US Army Corps of Engineers
Engineer Research and Development Center

CERL Technical Report 99/99
December 1999

CERL Noise Monitoring and Warning System 98

Daniel Sachs, Jonathan W. Benson, and Paul D. Schomer

The present CERL Noise Monitoring and Warning System, designed in the mid-1980s, has difficulty separating blast noise sounds from wind-induced pseudo-noise. A new noise monitor was designed that would be more wind noise resistant and would use more modern electronics and methods than those available in 1985. This report documents the design and construction of this new noise monitor.

The heart of wind-noise resistance is a two-microphone array and special signal processing to identify and separate blast sounds from pseudo-wind noise. The results are quite encouraging. It appears that the new monitor improves the signal-to-noise ratio by about 10 dB. It is recommended that this monitor be transferred to the field by a demonstration validation program such as the Environmental Security Technology Certification Program (ESTCP).

Foreword

This study was conducted for the Environmental Division at Fort Drum, NY, under Military Interdepartmental Purchase Request 6MCER50063, Work Unit H16, "Noise Control." The technical monitor was Loren Zeilnhofer, ATZS-PW-E.

The work was performed by the Ecological Processes Branch (CN-N) of the Installations Division (CN) Construction Engineering Research Laboratory (CERL). Al Schwark, Chief of the Fort Drum Range Division, was very helpful in time and resources to actually field and test the blast noise monitor. The CERL Principal Investigator was Paul D. Schomer. Dr. Harold Balbach is Chief, CN-N, and Dr. John Bandy is Chief, CN. The technical editor was Linda L. Wheatley, Information Technology Laboratory.

The Director of CERL is Dr. Michael J. O'Connor.

Contents

List of Figures and Tables

Figures

Tables

1 Introduction

Background

The Construction Engineering Research Laboratory (CERL) Noise Monitoring and Warning System is designed to recognize and report environmental noise due to on-base firing of heavy weapons. It consists of microphones, a control unit, and a base station that can be shared by multiple monitors. The first CERL Noise Monitoring and Warning System was built in the mid-1980s for installation at Fort Richardson, AK. Since then, many monitoring systems have been built and installed at different sites. However, the design has some technological limitations and relies on several parts, such as Hayes 300 bits per second (bps) modems, which are no longer in production. In 1996, CERL began to redesign the noise monitoring system using newer technology, addressing the limitations inherent to the older design. This work culminated in 1998 with the completion of testing of the prototype noise monitoring system delivered to Fort Drum, NY, in August 1997.

The older noise monitor units have several problems, most of which relate to the fact that the devices are quite simply very old. It is now difficult to acquire new parts with which to build new monitors, or repair the old ones; many of the parts have been discontinued, and several others are custom-built and therefore require many man-hours for assembly. In addition, the old control and signal processing hardware did not allow for much “intelligence” – the control software was very simple, with a primitive debugging interface and few configurable options. Since the design of the original noise monitors, advances in hardware, software, and signal processing algorithms have allowed these restrictions to be addressed.

Objective

The objective of this research was to test a new drop-in model of the CERL Noise Monitoring and Warning System to determine if it provides greater immunity from wind noise than the older model. The system is designed to inform an installation (typically range control) when impulse noise in the community or in other critical areas exceeds thresholds established by the installation. To do this,

a system consisting of a base station and one or more field units in locations selected by the installation is used.

For the older units, the menu-driven base station software controlled the unit and allowed all user parameters to be viewed and changed. Because the redesigned field unit is more versatile but adheres to the same interface as the older units, not all of the enhanced functionality is available from the base station software. This requires additional setup steps not applicable to the old monitor. However, the base station remains the central controller for an array of monitors, including both the old and new design.

The theory behind the redesigned noise monitor remains the same as the original. Its redesign was simply to take advantage of research in blast noise detection and more powerful computing hardware that has become available in the past few years. The redesigned field unit is a drop-in replacement for the older design. It replaces the older design, maintaining the same functionality, while using modern digital signal processor (DSP) hardware and specialized blast-detection software to improve noise immunity and allow blasts that are barely above wind-noise peaks to be detected and reported accurately. Like the old noise monitors, the new design incorporates a wind meter to warn of events that may be caused by wind noise; however, unlike the old monitors, internal software will reject most wind. The analog hardware for the new monitor provides a signal-to-noise ratio of approximately 65 to 70 dB, and a theoretical dynamic range of 96 dB. It is possible that improved analog design could improve the signal-to-noise ratio to approximately 85 dB.

Approach

A two-step approach was used to redesign the CERL Noise Monitoring and Warning System. The first step was to design signal processing algorithms to distinguish true blast noise from wind noise, which is the most significant source of false blast data collected by the previous monitors. The second step was to implement these new signal processing algorithms in a new field unit designed to improve upon the old field unit while remaining as compatible as possible.

For the new design, researchers used as much off-the-shelf hardware as possible, while still designing a functional and environmentally robust monitoring system. Therefore, the redesigned monitor was implemented primarily using standard components that included an embedded IBM-compatible personal computer (PC) controller and an off-the-shelf Industry Standard Architecture (ISA) DSP board. The few custom components are mostly slight modifications of the equivalent

components for the original field units. Although some of the individual parts are more expensive than the equivalent parts for the original field units, both share a modular design philosophy and failing parts can be easily replaced.

Major Improvements

Blast detection and filtering

The CERL Noise Monitoring and Warning System 98 incorporates hardware and software that helps prevent wind noise from being presented to the base station as blast data. The redesign includes two microphones and a DSP board running specialized blast-detection software, which allows the noise monitor to reject over 95 percent of wind noise while still recognizing genuine blasts less than a decibel above the wind noise floor. Using the DSP software and two microphones, the redesigned noise monitor substantially reduces the amount of invalid data taken while still recording any blasts that actually occur. The increased noise immunity allows the data-recording threshold to be lowered, thereby decreasing the false positive rate. With fewer false positives, the amount of data collected decreases, which reduces the cost of sending the data.

Higher transfer speeds

The new monitor also lowers telephone costs by supporting faster transfer rates between the field unit and the base station. Old CERL noise monitors used 300 bits per second (bps) modems, which took several seconds to transfer an acoustic data block. With new software and hardware at the base station, the new noise monitor can transfer an equivalent data block in less than a second. This reduces the amount of time the monitor spends online to a base station, further reducing cost of operation.

Outgoing call control

The new CERL noise monitor also adds the capability of controlling which data blocks result in outgoing calls. The original noise monitor placed calls when any data block was collected. If the noise monitors were used for real-time warnings, a large telephone bill resulted. The new noise monitor is capable of identifying data blocks that meet the users' criteria for importance, and only makes outgoing calls to warn of blocks exceeding these thresholds. Any other data are held in the monitor until they are collected by the base station.

Improved maintainability

The CERL Noise Monitoring and Warning System 98 also adds features that improve the maintainability of the hardware and firmware. Many operations that once required physical access to the unit, including upgrades to the internal control programs, can now be performed remotely via the onboard modem. Standard PC laptops running terminal emulation software can perform maintenance either directly or via modem; no console serial card or other extra hardware needs to be added to the monitor. Most diagnostics can be done without even opening the unit's case. In addition, the majority of the hardware components are industry standard, so it will be relatively easy to substitute different parts if a particular part becomes difficult to acquire.

Major System Changes From Original Implementation

The design of the new noise monitor unit is conceptually similar to the old units, but in a practical sense, they are very different. The old units were based on a CIM-100 embedded controller, and specialized software running under a multi-threaded version of FORTH (called SEVENTH); it also used a 68000 as a signal processor chip, running assembly code to do the summation for the true root mean square (RMS) calculations. This re-implementation of the monitor uses a similar overall structure: it consists of an i486-based controller card (running MS-DOS* and a C/C++ control program) and a dual Texas Instruments (TI) TMS-32C040 signal processing card. This card includes the analog-to-digital (A/D) converter. It finds peaks and total energies and performs digital-domain C-weighting and blast detection.

Despite architectural similarities, however, the two units have many differences. First, improvements in integration and Very Large Scale Integration (VLSI) design techniques since the design of the original monitor have enabled much more processing power to be put inside the monitor. The extra processing power offered by the i486 controller allows us to run a much more sophisticated control code. The extra signal processing power in the 32C040 chips allows the C-weight filter to be moved into the digital domain, resulting in improved accuracy and thermal stability. It also allows performance of blast-noise discrimination, which lowers the effective noise floor by as much as 5 dB.

* MS-DOS = Microsoft™ Disk Operating System.

The single largest architectural difference between the two units is the use of two microphones instead of one. The additional microphone provides two important advantages. First, it allows improved signal-to-wind-noise ratios by using the cross product of the two microphone's outputs, instead of the square of a single microphone. This method tends to cancel the wind noise, which has less correlation between the two inputs. The cross-correlation of the two microphones can also be used as part of the blast detection algorithm. Also, by using two microphones, some exploitable redundancy is gained. The loss of one microphone would compromise the ability of the monitor to discriminate blast noise from wind noise, but usable data can still be collected with only one working microphone.

Mode of Technology Transfer

It is recommended that the new CERL Noise Monitoring and Warning System 98 be transferred to the field via a suitable demonstration validation (DEM-VAL) program such as the Environmental Security Technology Certification Program (ESTCP).

2 Design Overview

Hardware

On the exterior, the new field units look much the same as the older noise monitor design. It uses the same environmental enclosure, air handling, filters, and many of the same connections. Internally, however, the layout is very different. The ISA bus backplane is shaped differently than the older CIM bus, and requires more careful ventilation. Unlike the old military-specification CIM cards, the ISA cards used in the new field units are commercial components designed for operation over a restricted temperature range of approximately 0 to 40 °C. To account for these hardware differences, the layout inside the monitor unit has been completely changed.

The redesign moves the card cage to the right side of the box, increasing airflow over the processing elements (see Figure 4 on p 75). On the left side of the box is a 1000-watt fan/heater used when the internal temperature becomes dangerously high or low. Although the 24-volt board from the older design was reused, its mounting and the mounting for the 24-V alternating current (AC) transformer had to be moved to make room for the ISA backplane on the right side of the field unit. Also, the modem holder at the top of the unit was replaced with a strap for the amplifier and power supply unit for the Larson-Davis (Provo, Utah) outdoor microphones. The main power supply for the field units was replaced with a conventional PC-type power supply with no battery backup, as the original monitor power supplies could not be reused. There simply was no room in the prototype unit for battery backup hardware, although further redesign could possibly add a battery backup.

Most of the field unit hardware components are commercially available industry standards: the outdoor microphone system, wind sensor, ISA card cage and power supply, and the ISA cards. Of the custom-built components, almost all use printed-circuit boards originally designed for the original monitor units (although the new audio input boards bear little practical resemblance to the original designs). The debouncer circuit that was added to solve last-minute problems is a small, wire-wrapped board. As stated earlier, use of industry-standard components will make replacement parts easier to acquire than parts for the older monitor units, which use several components that have been discontinued.

The new hardware also offers much more computing and signal processing power than the original monitor design, allowing the performance improvements.

The field unit also includes line conditioning hardware for both the incoming AC power and telephone line, which have effectively prevented power surges and spikes in incoming lines from damaging the field unit's internal circuitry.

Field Unit Controller Software

The field unit's primary controller is an Intel 80486-based single board computer running MS-DOS. It controls the ISA backplane and all other devices inside the field unit, and is responsible for all initialization, communication, and control tasks. The basic architecture of the field unit control software is a cyclic main loop, which rotates through a few basic tasks required to run the field unit:

- Check for new data from the DSP board, and accumulate and store any blocks collected.
- Check for new data from the modem, and send any pending data.
- Check for commands from an attached console keyboard or a serial-port pseudo-console.
- Check environmental data (such as internal temperature and system voltage), and turn the heater and fan on or off as appropriate.
- Check the event queue for other sporadic tasks (e.g., calibration sequencing and manual data block collection).
- Write data stored in internal data buffers out to disk.

In addition to the tasks called by the main loop, a few other tasks are performed by the monitor in response to internal and external interrupts. An internal clock, which triggers an interrupt 18.2 times per second, is used as a timing reference by the field unit software. The chopper-type wind meter also triggers an interrupt on every edge. This interrupt is used to count the number of times the wind meter makes a low-to-high transition. These data are combined with timings provided by the clock to provide an exact wind speed. Also, any data sent to or received from the serial port is buffered and handled by an asynchronous serial communications driver.

The field unit software is rather complex, but it has also proven to be fairly robust. Problems invariably crop up, however, in any actual implementation of a complex software program, so a hardware watchdog was included. This watchdog assures that, if the main loop of the host software ever fails to execute, the entire monitor will be restarted. If this happens, the monitor will be back up and taking data within a minute. Redundancies in the data storage procedure

ensure that, under normal circumstances, if the monitor is restarted, only one or two recorded data blocks at most will be lost.

Field Unit DSP Software

The field unit also contains a Spectrum Signal Processing DPC40B board, with two 40 MHz TI TMS320C40 DSP microprocessors, a Crystal Semiconductor ADC and digital-to-analog converter (DAC) board, and dual-port random access memory (DPRAM) to allow communication with the host PC. The two onboard DSP processors are identical, except that only the first processor can access the ADC and DAC board and the DPRAM. The two processors communicate via high-speed, buffered communications ports integrated onto the DSP microprocessors.

The field unit's signal-processing software has two primary processes, one running on each DSP. The first processor takes the data from the ADCs, performs a C-weight filter on each channel, and records the peak and sound exposure level (SEL) data. The flat and C-weight peaks and integrals, as well as those for the cross products of the two channels, are accumulated in tenth-second blocks and then made available to the host software. This greatly reduces demands on the host processor.

The second processor handles the blast detection process. Acoustic data are transferred from the first processor to the second. The second processor examines its input signal, looking for acoustic energy in frequency bands characteristic of blast noise. This information, as well as tests for correlation between the data acquired from the two channels, is used to identify blasts. The blast detection flags are then reported to the first processor, which accumulates and stores them with the rest of the data for each tenth-second block.

Base Station Software

The base station software used by the CERL Noise Monitoring and Warning System 98 is almost identical to the software used by the original field units. In fact, the field unit's control software is designed to emulate the original noise monitoring system's field units as closely as possible. This allows the new field units to be used interchangeably with the older units. The same base station software is used for both types of field units. A minor update was required to the field unit software used for the old monitor. The update adds support for some extra flags passed back by the new field unit.

The field unit also supports a special console mode that allows extended parameters (only applicable to the new field unit) to be set and field upgrades to

be performed. This console mode cannot be activated with the existing base station software. However, since the console mode simply uses standard American National Standards Institute (ANSI) terminal emulation, any communications software can be used. KERMIT, a free communication program distributed by Columbia University, is particularly well suited for this application. It supports ANSI terminal emulation, as well as the file-transfer modes used for the field upgrade process.

3 Noise Monitor Operation Manual

Base Station Configuration and Operation

From the perspective of the base station, the new field units are equivalent to the old design. They use the same protocols and transmit the same data. However, a minor update to the base station software is necessary to prevent the field unit's use of extra bits in the "windy" flag from causing unexpected effects. (Without this update, any event that did not trigger blast detection or that failed to pass the filter would be marked as windy.) With no significant updates to the base station code, base station operation is nearly identical. The only noticeable change is that the monitor will place fewer calls if the filtering features are enabled.

Base Station Hardware

The CERL Noise Monitoring and Warning System base station consists of a 286 or greater, IBM AT-compatible PC with a hard disk, and an attached modem, keyboard, and monitor. Optionally, a nearby Novell (Orem, Utah) server will accept all of the data collected by the base station (including both noise and weather data) and make it available to all other machines on a local network. Additionally, the base station is not particularly selective about the software that it runs on. This means that it could even be run as a background process under Windows® 95 or Windows® NT on a machine that serves as a personal workstation as well.

Some minor hardware differences exist between an ideal base station for the old units and the base station for the new units; for practical purposes, however, the two are interchangeable. The old field units have 300 bps modems, as did the original base stations. However, the 300 bps transfer rate is not sufficient for the console mode, so a faster modem (2400, 9600, or 14400 bps) is recommended for base stations controlling the new monitors. Also, the original base stations included a "listen mode" module, and an amplifier and speaker used to listen to the microphone on the field unit. "Listen mode" is not supported in the monitor redesign, so this hardware is useless for the new monitors.

A few requirements for the base station software need to be considered if off-the-shelf hardware is being assembled into a noise monitor base station. First, the base station control program is a DOS-only program designed to run full-time. It is best if it can run under DOS without Windows — this improves stability and decreases the likelihood of problems in the case of power failures. It also allows the use of either a timer or a software reboot program to reboot the base station on a regular basis. Second, the base station software only supports the COM1 and COM2 ports, so the modem must be attached to one of these two COM ports. If it is placed on COM3 or COM4, the base station software will not function. It is also best to use an external modem. The modem lights are useful diagnostic tools, and it is generally easier to assign COM1 and COM2 to the motherboard serial ports of modern machines than it is to assign them to a "Plug and Play" internal modem.

In addition, new modems must support 300 bps reliably. Not all high-speed modems reliably connect at 300 bps, especially when the modem at the other end is (as is the case for the new noise monitoring unit) also a high-speed modem.

Base Station Software

Two programs are used on the base station to control the new CERL field unit prototype. The first piece of software is a slightly modified version of the base station software for the original noise monitors. This version of the base station software is fully compatible with the original noise monitor software, and supports the two extra flags returned by the field unit: the blast detection flag and the filter flag. The second piece of software used with the new CERL field unit is a conventional terminal emulation program such as KERMIT, Telix, or Pro-COMM Plus. The terminal emulation software is used to access the field unit's maintenance console mode.

The base station software installation is fairly simple. If DOS (not Windows® 95) is already installed on the base station, and there is no Novell server or other networking required, the base station code (available on a floppy disk from CERL) must be copied into a directory on the base station's hard disk drive, and the "MON_330.EXE" program must be run from the AUTOEXEC.BAT file of the computer. If the computer has Windows® 95 installed, it should be replaced with DOS. (Although the base station program works under Windows® 95, CERL's system does not support that configuration.) If a prior version is installed, the new code (specifically, MON_330.EXE, MNU_330.TXT, and ERR_330.TXT) must be copied onto the base station, and MON_330.EXE must be executed from the base station's startup code.

If data collected from the field units need to be placed on a Novell server to be accessed from other computers, a few extra steps are required. First, the base station computer must be configured to access the network upon boot. Once this is done, the directory C:\FIRE must be created, and the file FIRE.CFG placed in it. This file needs to contain the line

NOISE MONITOR DATA=<directory for Noise Monitor data>

where the directory specified is a network drive. With this file in place, the base station software will automatically write a duplicate copy of all data it collects onto the network drive. Note that the above line should be added to any FIRE.CFG file already present, either in C:\FIRE or in F:\FIRE (the base station software, like CERL's other DOS tools, assumes that F:\ is the first network drive).

In addition to the base station software from CERL, a terminal emulator program is also required. CERL will supply a copy of KERMIT, a free DOS program that supports the console mode operation of the new field units. KERMIT can also transfer files to and from the field unit, as part of the field upgrade mechanism. To install KERMIT, simply copy the files from the distribution disk to a KERMIT directory on the hard disk of the base station. To use it, run the KERMIT.EXE program file, and use the following commands to set up the connection:

SET PORT <port number for modem>
SET SPEED <speed>
CONNECT

Next dial using the modem command "ATDT <phone number>". Note that, if a 14,400 or 28,800 bps modem is used, the SPEED should be 38,400 bps. If the modem is 2400 bps, use 2400 bps, and 300 bps modems should set a SPEED of 300 bps. Note that "console mode" on a 300 bps modem will be very slow. To access the console mode once the modem is connected, hold down CTRL-A until the status line appears. (Note that the status line can be disabled. If this has been done, check to see if the field unit has entered console mode by trying to type a command. If the typed characters appear on the screen [echo], the field unit is in console mode.)

Base Station Operation

Although it cannot control all aspects of the new field unit's operation, the base station software does perform many important functions. It is responsible for

collecting all data gathered by the field units (both new and old), setting all the parameters for the original field unit's operation, and setting all the new field unit's parameters that also exist in the older unit.

The base station software performs the following tasks:

- set the time of day
- Set the flat-weight, absolute threshold for data. (It can only set the "ABS_THRESH" option, not the other thresholds. Normally, this will not be used in the newer field units.)
- Set the wind-speed threshold, timeout, and sample interval, and whether or not windy data should be suppressed. (The console interface offers considerably more flexibility for these options.)
- sampling the noise level at a given field unit
- sampling the calibrator level at a given field unit
- enabling or disabling the accumulation of acoustic data at a given field unit
- listening to an unprocessed audio signal from an original field unit (This mode is not supported in the new units.)

The base station software is menu-driven, fairly straightforward, and easy to use. It has not changed significantly since the manuals for the older field units were written, so refer to these manuals for further information.

Field Unit Hardware Installation

The basic requirements for installation of the redesigned field units are as follows:

- standard voice telephone line
- standard 120-V AC service (60 Hz)
- 10-m tall pole
- two 2-m crossarms; one at the top of the pole, and one approximately 3 m lower
- one microphone and the wind meter are mounted on the top crossarm
- the other microphone is mounted on the bottom crossarm.

The field unit consists of the following parts:

- main case
- top microphone
- wind meter (attached to top microphone)
- bottom microphone.

The first step of field unit installation is to physically mount all components on the pole as shown in Figure 1. The field unit itself should be mounted low on the pole (but about 10 ft in the air, so it is less vulnerable to vandalism); the top microphone (which is labeled "1" and attached to the wind meter) should be mounted on one end of the top crossarm, and the bottom microphone ("2") should be mounted directly under it on the bottom crossarm. The wind meter should be mounted on the far side of the top crossarm.

To connect the field unit, bring the AC power into the nipple (installed on the right side of the bottom cutout) to the three screws on the left side of the isolation module (attached to the bottom right area inside the field unit, under the fan). Several other attachments are already there; do not remove them. The telephone line should be brought into a rubber grommet on the left side of the bottom cutout and attached to the right side of the telephone line isolator (attached under the audio input module). The AC power for the wind meter's heater should be brought into the AC power nipple as well, and then plugged into one of the two AC outlets mounted under the switch to the right.

Finally, the two microphone cables must be attached. Cable "1" (the top microphone) should be attached to the port labeled "1" on the bottom of the cutout, and cable "2" attached to the port labeled "2."

Figure 1. Field unit installation at Fort Drum, NY.

Finally, the monitor should be powered up. When the AC power is turned on, the light on the power supply in the center of the field unit will glow, and its fan will start (if it does not, check to make sure that the power supply is turned on). After approximately 30 seconds, the yellow light-emitting diode (LED) mounted on the 24-V board (attached to the top right corner of the case) will begin to flash on briefly approximately once every second. (This means that the internal control program is running, but data collection has been suspended.) After 3 to 4 minutes, the monitor will attempt to calibrate the microphones — the red lights on the audio module (labeled "1" and "2") will come on to indicate the calibrators are active.

Immediately after setup, the monitor should be attached to a laptop and tested. To do this, attach the laptop to the console serial mode, calibrate the microphones, take a few data points, and observe the result.

Field Unit Configuration

After the field unit is physically set up, it must be configured. This can be done using the unit's "console mode," described below.

Field Unit Console Mode

Because the feature set of the new monitors was completely redesigned without making major changes to the base station software, many of the new features can only be enabled using the console mode. Console mode is used to check out the monitor immediately after installation, and to set parameters and reconfigure the unit as it runs in the field.

The field unit console mode can be activated in two ways. First, a serial terminal (normally, a laptop with a communications program such as Telix or ProComm Plus) can be plugged into the console serial port in the monitor. If this method is used, the laptop should be set for 5700 bps, no parity, 8 data bits, and 1 stop bits. A null-modem serial cable should also be plugged into the 9-pin port on the left side of the ISA card cage.

Alternatively, an ISA video card and monitor can be plugged into the empty slot in the card cage, and a keyboard plugged into the keyboard connector. A monitor and keyboard attached in this way will serve as the console. (This connection is necessary for operations that require access to the DOS command prompt. The field unit software does not allow direct access to DOS via a serial port; however, most operations can be performed using KERMIT.)

If the physical field unit is not easily accessible, the console mode can also be accessed via the internal modem. To do this, dial up the field unit with a standard terminal program. After it connects, hold down Ctrl-A for a few seconds. After the field unit has received 10 consecutive Ctrl-A characters, the unit will go into console mode. At this point, a status line will appear at the top of the screen (unless it has been disabled; see the command summary under Base Station Software).

Console mode allows direct access to almost anything the monitor can do. It can be used to view (with more detail) and delete any data that the monitor collects.

Appendix A contains a list of commands available in the console mode. However, certain console mode commands are particularly important:

SHOW: Shows information on blocks stored in the queue, thresholds, filter settings, or option settings.

HELP: Gets help. HELP <command> displays the help file for a command; HELP SET <option> displays help for an option.

SET: Sets an option. SET <option> <parameters>.

CAL: Take a calibration block.

HANGUP: Hangs up the phone. (Use to disconnect a remote console mode.)

START: Take a manual data block. Normally used as "START <time>" to take a <time> second data block. The results are stored and displayed on the screen.

Field Unit Configuration

The field unit has a large list of configurable options. Many of them have usable defaults or can be set from the base station; however, because the base station software and protocol were designed for the original noise monitor unit, several important parameters can only be set from the console mode. The console mode command to set a parameter is:

SET <parameter> <option>

Many parameters are available; Appendix B is a complete list. Following is a summary of important settings that can only be made in console mode. These

should be set as part of the field unit setup procedure. The "*" refers to everything that starts or ends with the characters around it.

CAL_LVL: Set the calibrator level of the microphones. Should be set to correspond to the actual acoustic energy (dB) that corresponds to the calibrators. (This level is only used for the console displays. In future versions of the field unit and base station software, it may be used for actual data as well.)

*_THRESH: Parameters ending in _THRESH control the data collection threshold. The monitor will accumulate data (starting PRETRIG seconds before the threshold and continuing POSTTRIG seconds afterward) if a tenth-second block exceeds all of the set thresholds.

*_FILTER: Parameters ending in _FILTER control the internal filter. This can be used to reject data blocks or to identify blocks as being "important." The field unit will only call the base station when its memory holds more than a certain number of blocks (CALL_THRESH).

KEEP_*: The original noise monitor allowed blocks to be accepted or rejected based on wind speed. The newer noise monitors allow for more extensive control over conditions under which a block will be kept, and whether or not a warning call will be made. These options are controlled by KEEP_BAD, KEEP_WINDY, and KEEP_FILTER.

CALL_THRESH: Unless an "important" block comes in, the field unit will not call the base station until there are at least this number of blocks in the queue.

DATE: Sets the field unit time. (SET TIME hr:min:sec)

TIME: Sets the field unit time. (SET DATE month/day/year)

(Note that the date and time can be set with the base station software as well.)

Finally, having two microphones requires that the base station's calibrator constant be set to the average of the calibrator constants of the two microphones. (A

later release of the field unit and base station software may have better support for the two-microphone configuration.)

Data Collection and the Blast Detection Algorithm

One important new feature added in the field unit redesign was a blast detection algorithm. This algorithm is described in more detail later; however, properly setting up the threshold for the field unit requires an understanding of how the blast detection algorithm operates.

The SEL and peak level data provided by the monitor are based on the raw data from the microphone (sampled at a rate of 48,000 samples per second), while the C-weight SEL (CSEL) and C-weight peak data are based on the microphone data after they are passed through an implementation of a C-weight filter. The data are accumulated into tenth-second blocks, and the following data are stored for each block:

- peak value seen for each microphone, and the position of this peak within the tenth-second block
- peak value seen for the C-weighted data from each microphone, and the peak's position
- peak value seen for the cross product of the data from the two microphones, and its position
- peak value seen for the cross product of the C-weighted data, and its position
- the sum of the squares of the microphone samples for each microphone
- the sum of the square of the C-weighted data for each microphone
- the sum of the cross-microphone products for the flat-weighted and C-weighted data.

All these data are passed to the controller program, which normally stores the cross product data for later transmission to the base station. (This behavior can be modified using the CHANNEL_MODE option.) The thresholds and filters can be set on either flat or C-weighted data; however, only the flat weight peak and C-weight SEL data are passed on to the base station.

The raw 48 kHz data are also decimated to 2 kHz; the remaining samples are then passed on to the blast detection algorithm. After receiving each sample, the blast detection algorithm decides whether it is likely that a blast is currently in progress. This determination is based on two factors:

- The energy in a frequency range characteristic of blast noise (approximately 10 Hz to 35 Hz) must be at least 25 percent of the largest energy in that band seen in the last third of a second, not including the last 35 ms.

- There must be at least a 70 percent cross-correlation between the two channels. (Wind will tend to have a lower cross-correlation.) This will cut out much of the wind noise, which tends to be poorly correlated between the two microphones.

If both of these conditions are met, the incoming sample is considered to be part of a blast. When this happens, a blast detection "blip" is passed back and included with the tenth-second block that is currently accumulating. The total number of blast detection blips in each tenth-second block is stored with the block. The number of blips in a given block can then be compared with a threshold to determine if there is, in fact, a blast event in progress.

If BLAST_THRESH (this threshold) is set to 1, the blast detection algorithm will be able to identify blasts at 0 dB relative to wind noise, and have a false positive rate that allows approximately 3 percent of non-blast peaks to be stored. Normally, this blast detection threshold will be set to 1. However, because raising the threshold somewhat decreases the false positive rate, it may be useful in noisy environments. This decrease in the false positive rate, however, comes at the expense of somewhat increasing the minimum signal-to-noise ratio required to trigger the threshold. Using a blast threshold of 7 (representing 7 blast detection flags out of the 200 possible in each tenth-second block) seems to allow most "real" blast data to be recorded, and minimizes wind noise.

The blast detection algorithm has been extensively characterized for the case where the blast threshold is one blast-detection "blip." (These data are included in Chapter 4.) However, it has not been characterized for higher numbers of detections, and little empirical performance data exists. Also note that it is possible for the blast detection blips generated by the blast's peak to be split between two adjacent tenth-second blocks, so using high value for the blast threshold may result in unrecorded blasts. Although it is known that using a higher number for the blast threshold increases wind-noise immunity and sensitivity to weak blasts, it is not known exactly how much these two values change. Therefore, if the blast threshold is set to a value greater than one, the user should verify that valid blast data are not being ignored.

Thresholds, Filters, and Data Collection

An important part of setting up the monitor is setting the threshold and filter options. Unlike the old noise monitoring units, which had only one threshold option (flat-weighted peak), the new monitors support a variety of trigger threshold parameters. Data collection will be triggered if, and only if, all the thresholds are met simultaneously.

Threshold basics

The original CERL noise monitor units had only one threshold to set — they compared the absolute value of the signal from the ADCs to this threshold, and triggered data collection whenever this threshold was exceeded. For compatibility, the redesigned units preserve the absolute threshold. However, to take full advantage of the new design, additional trigger conditions were added. In practice, these new conditions should be used — the absolute threshold can be left at a very high value, or turned off completely.

By setting the proper thresholds, the monitor can be programmed to trigger when a variety of different conditions occur. Data blocks are taken when all of the trigger conditions are simultaneously satisfied by a tenth-second block. To use the monitor to detect blast noise, set the CPK_THRESH (C-weight peak acoustic level) and FPK_THRESH (flat-weight peak acoustic level) as well as BLAST_THRESH (which controls the blast-detection algorithm sensitivity). For other applications, the CSEL_THRESH (C-weight tenth-second SEL) and FSEL_THRESH (flat-weight tenth-second SEL) thresholds can also be set.

These thresholds can be applied to either microphone or a composite of both. If both microphones are enabled (which is the default state), these thresholds will be compared against the cross peak and cross SEL values measured by the monitor. These cross values are obtained by assuming the product of the two microphones' outputs is the square of the "actual" measurement. In laboratory testing, this method of measurement was found to improve the wind immunity of the system without measurably affecting the peak and SEL levels seen by the monitor. When in the dual-microphone mode, the cross values used for the threshold are also the values reported to the base station. In single-microphone mode, the peaks and SELs seen by that single microphone are sent.

For example, at the Fort Drum Sprague ville test site, the following set of thresholds were used:

```
SET CPK_THRESH 95
SET FPK_THRESH 100
SET BLAST_THRESH 1
```

These settings mean that, for a trigger to occur, all of the following conditions must be satisfied:

- The observed C-weight peak for a tenth-second block must be at least 95 dB
- The observed flat-weight peak for a tenth-second block must be at least 100 dB

- The blast-detection algorithm must have detected a blast.

These values were chosen based on the observed noise floor of approximately 100 dB in light to moderate winds. A 100-dB blast will result in an observed C-weight peak of approximately 95 dB; because the C-weight filter removes 10 to 15 dB of wind noise and only about 5 dB of blast noise, the combination of the C-weight and flat-weight thresholds improves the wind noise immunity of the noise monitor substantially.

An even larger improvement can be achieved by setting the blast detection threshold to 1 or more. A threshold of 1 is the "normal" value, and results in a very low incidence of undetected blasts. Higher blast detection threshold values result in even better wind noise immunity; however, more blasts will go undetected as the blast threshold rises. The exact effects of values greater than one on the blast detection algorithm have not been quantified; however, if the blast threshold is set to greater than 50, only the very strongest blasts will be stored. Any value greater than 200 will completely disable data collection.

Available thresholds

The monitor triggers data collection when all of the enabled thresholds are satisfied by the same tenth-second block. This trigger accumulates data PRETRIG seconds before the trigger tenth-second block and POSTTRIG seconds after, and forms a data block that is sent to the base station. Any thresholds may be disabled with the command "SET <threshold name> OFF."

ABS_THRESH (absolute thresholds) corresponds to the threshold for the old monitor. ABS_THRESH is compared against an absolute value coming off the ADCs. Any changes in the gain constant of the microphones will affect the acoustic decibels that this threshold corresponds to. This threshold option was included to maintain compatibility with the older field unit design. However, it serves the same purpose as the acoustic level thresholds so it will normally be left unset.

Because ABS_THRESH is intended only for compatibility with the base stations, normally it will be set by the base station: "Call a Unit" / "Display/Modify Unit Parameters" / "Peak Threshold." However, it can also be set using console mode. Use "SET ABS_THRESH <1..32767>" to set it as a number (with 32767 = 96.3 dB relative to the minimum detectable sound level, and 1 being the minimum detectable sound level). It can also be set in decibels using "SET ABS_THRESH <number> dB" (where 96.3 dB is the maximum possible), or in calibrated, acoustic sound levels using "SET ABS_THRESH <number> ABS dB."

FPK_THRESH and ABS_THRESH are similar; however, FPK_THRESH stores an acoustic decibel level. To work properly, the calibrator values must be set properly. Whenever ambient noise peaks exceed the given level (in decibels), this threshold will be triggered. The FPK_THRESH variable is set using "SET FPK_THRESH <level>" where <level> is the acoustic threshold level in decibels.

Likewise, CPK_THRESH triggers whenever C-weighted ambient noise levels exceed that level. C-weighting the peaks is useful, because much of the wind noise is blocked by the C-weight filter, while much of a blast is passed. To set the C-weight peak threshold, use "SET CPK_THRESH <level>."

FSEL_THRESH and CSEL_THRESH are triggers based on the total energy contained in tenth-second blocks. Note that these values do not correspond to the total SEL over an entire 2+ second data block; the data block is not accumulated until after the trigger takes place. The filter can be used to limit collected data to only high total energy blasts. These values are set using "SET FSEL_THRESH <level>" and "SET CSEL_THRESH <level>."

BLAST_THRESH controls the blast detection algorithm. Turning BLAST_ THRESH off ("SET BLAST_THRESH OFF") disables blast detection completely. This can be useful for monitoring non-blast noise, but severely reduces the immunity of the monitor from false blocks due to wind noise. "SET BLAST_ THRESH 1" enables blast detection; the threshold condition includes a signal from the blast detection algorithm that a blast is in progress. Setting BLAST_ THRESH to 2 or more will further reduce wind noise but will decrease the likelihood of quiet or indistinct blasts being recognized as blast noise.

Data accumulation and collection

Once the thresholds are satisfied, the monitor begins to accumulate tenth-second blocks into a data block to be sent to the base station. The monitor retrieves the data for the last PRETRIG seconds (in tenth-second increments) from its circular queues, and continues to accumulate data for POSTTRIG seconds after the trigger event. If additional triggers occur during this time, the data accumulation proceeds until POSTTRIG seconds after the last trigger.

To set the pre-trigger and post-trigger accumulation times, use the "Call a Unit->Modify Unit Parameters" base station menu option or the following console commands:

SET PRETRIG <seconds>
SET POSTTRIG <seconds>

The default value for PRETRIG is 0.5 seconds, and the default value for POSTTRIG is 1.5 seconds. Note that, due to the finite size of the circular buffer used to store past tenth-second blocks, the pre-trigger time is limited to 4 seconds. The post-trigger time can be up to several thousand seconds.

The datum returned as PEAK is the highest peak seen in the entire block (including pre-trigger and post-trigger times), and the datum returned as SEL is the C-weight SEL across the entire block. Since the SEL accumulates all sound energy seen over the entire period, it is best to set the pre-trigger and post-trigger values only high enough to capture the entire blast. Wind noise can become a significant fraction of the total energy for a very large block, and the blast noise detection algorithms are only applied to the trigger. The C-weight filter rejects most of the wind noise, but what remains can still skew the detected blast energy.

Block filters

Once the data block has been accumulated for the post-trigger time after the last trigger, the final block is then filtered according to several rules. An acoustic filter "passes" a block based on acoustic criteria. In addition, blocks may be filtered based on the presence of a blast-detection flag somewhere in the block, and based on whether the "windy" flag was set any time during the collection of the data block.

Blocks that pass all of these filters are marked as "important." An "important" block triggers a call from the monitor to the base station; an "unimportant" block does not. Typically, this filtering will be used to separate low-level blast noise (which will be reported during a nightly call from the base station to the monitor) from loud blasts that should be reported immediately. In this way, the field unit becomes more useful as both a monitoring device and as a warning device. It collects data on low-level noise events without tying up the telephone line by calling every few minutes. A block with the "reject" flag is not recorded or sent to the base station.

Three separate flags control the actions of the filters: KEEP_BAD, KEEP_FILTER, and KEEP_WINDY. These interact with each other through two flags: "block accepted" and "block is important."

Initially, each block is marked as both accepted and important. These flags are cleared based on the conditions in the KEEP settings. If a KEEP setting is set to "CALL," the condition it controls will have no effect on either the "reject" flag or the "important" flag. If a particular KEEP setting specifies that a block should

be rejected or marked as unimportant, that takes precedence over any other setting indicating that the block should be kept or is important. All of the KEEP settings apply only if the condition they signify applies. The KEEP_BAD setting applies if the block is rejected by blast detection, the KEEP_WINDY flag applies if a block is marked as windy, and the KEEP_FILTER setting applies if the block does not satisfy the filter thresholds.

KEEP_BAD is the simplest of the KEEP settings. In fact, if the monitor is configured to trigger only on a blast detection (BLAST_THRESH is set to 1 or more), the monitor essentially ignores it because all blocks will contain blast detections. However, if BLAST_THRESH is set to 0 (ignoring the blast detection algorithm for the thresholds), this setting can be used to control the disposition of blocks that did not contain a detected blast. If KEEP_BAD is OFF, the block will be rejected. If KEEP_BAD is ON, the block is not rejected but the importance flag is cleared — the block will not result in a warning call to the base station. Finally, if KEEP_BAD is set to CALL, the block will be kept and the importance flag will not be cleared. Either way, the field unit will mark the block as "BAD" (no blast detection) when it is sent to the base station.

The KEEP_WINDY and KEEP_FILTER settings are more complex, but this complexity is masked if the monitor is configured to trigger only when a blast is detected. If KEEP_WINDY is set to CALL, the windy flag is passed to the base station but otherwise ignored. If KEEP_WINDY is ON and a data block is marked as windy, the importance flag is cleared. If KEEP_WINDY is OFF, a data block marked as windy is rejected. If BLAST_THRESH is set to 0, two additional settings apply: BLAST and CALL-BLAST. If KEEP_WINDY is set to BLAST, a block that is windy will be rejected if no blast was detected during that block. If a blast was detected, it will be kept with the importance flag cleared. If KEEP_WINDY is set to CALL-BLAST, a block with a blast detection will not be rejected and the importance flag will not be cleared.

KEEP_FILTER behaves exactly the same way. If it is set to CALL, the fact that the filter did not accept the block is passed on to the base station but otherwise ignored; if ON, the importance flag will be cleared; if OFF, the block will be rejected. Likewise, if BLAST_THRESH is 0, BLAST will not reject a block that contains a blast detection, and CALL-BLAST will not clear the importance flag for such blocks.

Unlike the thresholds, which are triggered only if *all* of the specified conditions are met, the acoustic filter passes a block if *any* of the conditions are satisfied. If the accumulated data block meets or exceeds any of these filter settings, the

filter will pass the block, and the conditions specified by KEEP_FILTER will not be applied. The acoustic filter supports four conditions:

FSEL_FILTER: The flat-weight SEL for the entire data block
CSEL_FILTER: The C-weight SEL for the entire data block
FPK_FILTER: The highest flat-weight peak for the entire data block
CPK_FILTER: The highest C-weight peak for the entire data block

All of these are set using the SET <filter> <decibel level> command, or cleared using the SET <filter> OFF command. A cleared filter will never pass any data. For example, in the test setup at Fort Drum, the following filter settings are set up:

```
SET FPK_FILTER 115
SET CPK_FILTER OFF
SET CSEL_FILTER OFF
SET FSEL_FILTER OFF
```

These settings pass any block that contains a peak of 115 dB or more. The KEEP settings are set to ignore wind speed and mark only blocks that pass the filter as important. The KEEP_BAD is irrelevant because the BLAST_THRESH option is set to 1. These settings are achieved using the following commands:

```
SET KEEP_WINDY CALL
SET KEEP_FILTER ON
```

The following truth table (Table 1) describes the interaction between the KEEP flags. Note that, if BLAST_THRESH is zero, then all blocks that are taken will be "blast" blocks, and the value of KEEP_BAD is essentially ignored. CALL indicates that the block is marked as important — the monitor will attempt to call the base station if one or more important blocks are in the queue.

Table 1. Interaction between KEEP flags.

KEEP_WINDY, KEEP_FILTER, KEEP_BAD	No blast Windy Below filter	Blast Windy Below filter	No Blast Not Windy Below filter	Blast Not Windy Below filter	No Blast Windy Passes filter	Blast Windy Passes filter	No Blast Not Windy Passes filter	Blast Not Windy Passes filter
CALL, CALL, CALL	PHONE	PHONE	PHONE	PHONE	PHONE	PHONE	PHONE	PHONE
CALL, CALL, ON	KEPT	PHONE	KEPT	PHONE	KEPT	PHONE	KEPT	PHONE
CALL, CALL, OFF		PHONE		PHONE		PHONE		PHONE
CALL, ON, CALL	KEPT	KEPT	KEPT	KEPT	PHONE	PHONE	PHONE	PHONE
CALL, ON, ON	KEPT	KEPT	KEPT	KEPT	KEPT	PHONE	KEPT	PHONE
CALL, ON, OFF		KEPT		KEPT		PHONE		PHONE
CALL, OFF, CALL					PHONE	PHONE	PHONE	PHONE

KEEP_WINDY, KEEP_FILTER, KEEP_BAD	No blast Windy Below filter	Blast Windy Below filter	No Blast Not Windy Below filter	Blast Not Windy Below filter	No Blast Windy Passes filter	Blast Windy Passes filter	No Blast Not Windy Passes filter	Blast Not Windy Passes filter
CALL, OFF, ON					KEPT	PHONE	KEPT	PHONE
CALL, OFF, OFF						PHONE		PHONE
ON, CALL, CALL	KEPT	KEPT	PHONE	PHONE	KEPT	KEPT	PHONE	PHONE
ON, CALL, ON	KEPT	KEPT	KEPT	PHONE	KEPT	KEPT	KEPT	PHONE
ON, CALL, OFF		KEPT		PHONE		KEPT		PHONE
ON, ON, CALL	KEPT	KEPT	KEPT	KEPT	KEPT	KEPT	PHONE	PHONE
ON, ON, ON	KEPT	KEPT	KEPT	KEPT	KEPT	KEPT	KEPT	PHONE
ON, ON, OFF		KEPT		KEPT		KEPT		PHONE
ON, OFF, CALL					KEPT	KEPT	PHONE	PHONE
ON, OFF, ON					KEPT	KEPT	KEPT	PHONE
ON, OFF, OFF						KEPT		PHONE
OFF, CALL, CALL			PHONE	PHONE			PHONE	PHONE
OFF, CALL, ON			KEPT	PHONE			KEPT	PHONE
OFF, CALL, OFF				PHONE				PHONE
OFF, ON, CALL			KEPT	KEPT			PHONE	PHONE
OFF, ON, ON			KEPT	KEPT			KEPT	PHONE
OFF, ON, OFF				KEPT				PHONE
OFF, OFF, CALL							PHONE	PHONE
OFF, OFF, ON							KEPT	PHONE
OFF, OFF, OFF								PHONE
BLAST, CALL, CALL		KEPT	PHONE	PHONE		KEPT	PHONE	PHONE
BLAST, CALL, ON		KEPT	KEPT	PHONE		KEPT	KEPT	PHONE
BLAST, CALL, OFF		KEPT		PHONE		KEPT		PHONE
BLAST, ON, CALL		KEPT	KEPT	KEPT		KEPT	PHONE	PHONE
BLAST, ON, ON		KEPT	KEPT	KEPT		KEPT	KEPT	PHONE
BLAST, ON, OFF		KEPT		KEPT		KEPT		PHONE
BLAST, OFF, CALL						KEPT	PHONE	PHONE
BLAST, OFF, ON						KEPT	KEPT	PHONE
BLAST, OFF, OFF						KEPT		PHONE
CALL-BLAST, CALL, CALL		PHONE	PHONE	PHONE		PHONE	PHONE	PHONE
CALL-BLAST, CALL, ON		PHONE	KEPT	PHONE		PHONE	KEPT	PHONE
CALL-BLAST, CALL, OFF		PHONE		PHONE		PHONE		PHONE
CALL-BLAST, ON, CALL		KEPT	KEPT	KEPT		PHONE	PHONE	PHONE
CALL-BLAST, ON, ON		KEPT	KEPT	KEPT		PHONE	KEPT	PHONE
CALL-BLAST, ON, OFF		KEPT		KEPT		PHONE		PHONE
CALL-BLAST, OFF, CALL						PHONE	PHONE	PHONE
CALL-BLAST, OFF, ON						PHONE	KEPT	PHONE
CALL-BLAST, OFF, OFF						PHONE		PHONE

Field Unit Maintenance

The field unit's hardware and software are designed to be robust and require little attention, but it will occasionally be necessary to perform maintenance to cor-

rect software flaws or fix broken hardware. The field unit is set up so that both hardware and software modules can easily be replaced in the field if problems are found. In fact, software problems can generally be rectified over the modem, without any onsite presence whatsoever.

Preventative Maintenance

Although the field unit is robust and designed for unattended operation outdoors for extended periods of time, it does require regular preventative maintenance to stay in good operating condition.

First, the field unit contains air filters — a steel filter and a foam filter on the left side (the air intake) and a steel filter only on the right side (exhaust). These filters, which are on the bottom of the unit, help keep insects and dust out of the field unit while allowing the unit some control over its internal temperature. These filters should be checked periodically — the foam filter should be replaced when it becomes dirty or clogged, and the steel filters should be washed. Failure to do so can interfere with internal environmental control and cause the unit to overheat.

Second, both microphones have desiccant capsules designed to keep the microphone cartridge dry. These capsules should be replaced periodically to prevent damage to the microphone. The Larson-Davis microphones are especially particular in this regard. The desiccant cartridges are too small to hold much silica gel, so they should be replaced every 3 to 4 months. If this is not done, the microphones are likely to fail. This is believed to be a contributing factor to the frequent microphone failures observed during testing.

Third, the field unit should periodically be checked and cleaned inside. All of the fans and the heater should be checked to ensure they are working (this can be done using the “SET FAN ON” and “SET HEATER ON” console-mode commands).

The field unit also contains two batteries (one powers the internal clock, the other powers the nonvolatile RAM storage) that should be replaced every 5 years. The clock battery is on the central processing unit (CPU) board; the RAM backup battery is on the PCF-1 disk emulation board.

Field Unit Service Mode

During testing, it often proved necessary to update the controller code stored in the read only memory (ROM) of the prototype field unit. Because it is not possible to write a new image into the field unit's boot ROM without physical access to the unit, it was necessary for this update to take place through a different mechanism. Therefore, the field unit's control code, as well as the DSP code, can now be upgraded remotely using the KERMIT in a special "field service" mode.

WARNING: Extreme care should be taken when using field service mode, as damage to the field unit may result from its improper use. When the unit is in field service mode, closed-loop climate control is disabled — the system's heater and fan will not run. Do not use the field service mode if it is hot or cold outside the monitor, as the unit may be damaged by temperature extremes. During the summer, use the field service mode only at night. Also, if the new code uploaded into the unit fails to boot, the monitor will need to be opened to be reset. In addition, if the controller code fails, neither the heater nor the fan will run. This will result in damage to the field unit's internal hardware.

Upgrading the field unit's control program using KERMIT is fairly straightforward. Start by using the MS-DOS KERMIT program to connect to the field unit. This can be done using the procedure described on page 21 for accessing the field unit console mode. Once the field unit has been placed into console mode, issue the command "KERMIT MODEM" to the monitor. This tells the field unit to shut down and enter service mode. Wait approximately 30 seconds, and then hit Ctrl-] followed by "c" to return to the KERMIT prompt.

Once you have the KERMIT prompt, test to make sure that the field unit successfully entered service mode by typing the command "REM DIR". This should promptly return an MS-DOS directory listing. If no directory listing appears within 15 seconds, hang up, wait approximately 2 minutes, and try again.

Upgrading the field unit's control software using service mode is fairly easy. It can be done by issuing the following commands from the KERMIT prompt:

SET FILE TYPE BINARY	These commands set up the KERMIT transfer parameters.
SET RECEIVE PACKET 1000	
SEND <local path to MAIN.EXE> D:\MAIN.EXE	This sends the new copy of the control program to

	temporary storage on the field unit.
REM HOST COPY D:\MAIN.EXE C:\MAIN.EXE	This copies the control program from temporary storage to permanent storage.

When this procedure has been fully executed, type “bye” to hang up and reboot the field unit. When the field unit comes back up, use the “VERSION” command to check the build date and verify that the correct version is running. (If the incorrect version is running, it is possible that an error occurred in transferring the code or in starting up the new code. If this happens, verify that the version being uploaded works properly and send it again.)

Although the DSP software and the boot procedure are not expected to need changing in the field, it is possible to change the DSP code and parts of the boot sequence using the field service mode. (It is recommended that changes to parts of the software other than the main control program be done by updating the monitor’s FLASH ROM. FLASH ROM is discussed in the next section.) Upgrading the DSP software or changing the startup sequence requires a slightly more involved procedure. To upgrade the DSP software, the updated DSP code files C:\MONDSP_1.OUT and C:\MONDSP_2.OUT must be loaded onto the field unit using the above procedure. Second, a file labeled C:\STARTUP.BAT must be created on the monitor, containing the following line:

```
SET DSP_PATH=C:
```

Finally, additions can be made to the startup procedure by including additional commands in the STARTUP.BAT file. This file is called by AUTOEXEC.BAT, and can be used to perform additional processing. When the STARTUP.BAT file exits, the normal field unit startup procedure will continue.

Field Unit FLASH Update Procedure

Occasionally, it will be necessary to upgrade the boot image that the monitor stores in its FLASH ROM. This is actually easy — as the field unit includes a self-update mechanism. Upgrading the FLASH ROM of a field unit requires two parts that are not part of the main field unit:

- a standard 3.5 in. floppy disk drive
- a standard PC-AT keyboard.

To upgrade the FLASH ROM on the field unit, first turn the field unit off at the power switch. Attach the floppy disk drive to the 34-pin floppy disk port on the field unit's CPU card (making sure to line up pin 1 at both ends) and to a power connector coming off the power supply. Also, attach the keyboard to the 5-pin DIN keyboard connector attached to the backplane. Then, simply insert an updated image disk into the floppy disk drive, and press the "3" key as the field unit is booting. The field unit will automatically read the image disk and store it into its FLASH memory. As it does this, the drive activity light on the 3.5 in. floppy disk drive will flash briefly, turn off, and then stay on for an extended period of time. When it turns off again, it is safe to turn off the field unit and remove the added hardware — the ROM update is complete.

In addition, it is possible to destroy all data stored in the field unit's nonvolatile RAM disk by pressing the "4" key on the attached keyboard as the field unit is booting. This allows for easy recovery from a failed remote upgrade attempt. Note that this deletes any stored data and configuration; all settings will be reset to the defaults stored on the image disk that was loaded into the monitor's FLASH ROM.

4 Noise Monitor Testing Results

The prototype CERL Noise Monitoring and Warning System 98 was built primarily as a test bed for several new techniques in noise monitoring and blast detection, in an effort to increase the wind noise immunity of blast noise detection systems. A secondary goal was to design and build a reliable noise monitoring system that could replace the older monitors, taking advantage of modern technology. As a result, the blast detection algorithm and the DSP software that implements it underwent extensive testing to prove that it is effective at rejecting wind noise and identifying blasts. In addition, the control software underwent extensive testing to weed out "bugs" and reliability problems in the laboratory before release to the field. Testing is now underway at Fort Drum, where the complete prototype field unit is serving as a noise warning station.

Laboratory Test Results for the Blast Recognition System

Benson (1996) provides more information on the blast recognition system. Results of blast/wind noise tests and measurement simulations are reported in this section.

Results of Blast Noise Detection / Wind Noise Rejection Tests

The performance of the blast detection and wind noise rejection algorithm was thoroughly tested in the laboratory. First, Monte Carlo simulation programs were designed to test the performance of the algorithm on randomly chosen segments of wind noise, or wind noise with a blast present. Real wind noise and artillery blasts were digitized audio recordings for these simulations. For each different wind noise recording, the peak value of the noise was found. A random number seeded from the computer's clock was used to select a point within the segment to insert a blast event. The blast was inserted with a given relative amplitude with respect to the peak wind noise value found. The segment of the wind file with the blast (including some lead and lag time surrounding the blast) was then operated upon by the discriminator algorithm. Records of the number of detections were tabulated. The false detection rate, defined as the percentage of instances where a blast is detected when there was no real event, was tested and tabulated by adding the blast into the wind noise at a relative level of -100

dB. For all practical purposes, the blast was absent. Several trials were performed for each of several different wind noise recordings.

Table 2 shows the results from several of the simulations. Two different blast signals were used in the simulations. The blasts differ in their spectral content, one having its energy peak at 25 Hz (typical blast) and the other at a low 7 Hz. The low frequency blast was used to test the discriminator's ability to detect blasts that may have traveled over long distances. Blasts incurring long propagation distances are rarely a noise problem, as their sound pressure levels are usually very low. They do, however, provide an interesting test of the discriminator's ability. The simulations helped to find an optimum performance balance between the ability to detect blasts and the ability to screen out events caused by the wind noise. Through the many trials, general trends were observed concerning the effect of each parameter on the performance of the algorithm.

Table 2. Blast detection probabilities (Monte Carlo testing).

Blast Energy Peak (Hz)	Blast Level (dB)	Total Trials	Percent Detected
25	3	750	99.2
25	0	1700	97.4
25	-3	1700	82.9
25	-6	1500	52.8
25	-10	1350	20.0
25	-100*	1700	0.12
7	3	750	98.5
7	0	1500	92.7
7	-3	1500	73.1
7	-6	750	47.5
7	-10	600	15.8

The "percent detected" values in Table 2 indicate acceptable performance. The algorithm is able to detect 97.4 percent of blasts that are present with an amplitude equal to the peaks in the wind noise surrounding the blast (blast level = 0 dB). This percentage raises to nearly 100 percent when the blast's level is 3 dB or more above the wind noise peaks. The false detection rate seems good at 0.12 percent. This value could be misleading, as it suggests that a false detection would occur only once every 14 hours or so. This is not the case (as discussed below). The detection rate falls off as the blast's level drops below the wind noise peaks. At -6 dB, over 50 percent of the blasts are still detectable. The detection rates for the very low frequency blast (peak energy at 7 Hz) are less than those for the normal blast. These results were better than expected, as the spectrum

of this low frequency blast has much less energy in the energy function band than the typical blast.

Second, several different tests were performed to evaluate the real-time performance of the discriminator. The audio tapes previously used as the source for the wind noise characterization work, were played back in real-time into the DSP-based discriminator. Also, further recordings were made under varying wind conditions to provide additional real-time input for the system. The discriminator was linked to a test bed, a two-channel version of the original CERL Noise Monitoring and Warning System. This linking allowed the gathering of data on the level of the wind noise and allowed real-time testing of the false detection rate of the discriminator.

When the algorithm decided that a blast was present, it sent a 1-V direct current (DC) signal to the monitor. The monitor was set up so that two conditions would trigger on an event. The audio signal sent to the monitor must exceed the set threshold, and the 1-V signal from the discriminator must be present. This setup allowed one to view the performance of a system that had the discriminator included as part of the threshold condition for trigger. To compare the results to that of a standard noise monitor, the same data tape could be played into the monitor without the discriminator controlling the triggering. Many different wind noise recordings were used as real-time input into the monitor/discriminator combination. Table 3 compiles the results. On average, the discriminator was able to reduce the number of false events by 97.5 percent. This reduction is significant and useful. The results do not show any strong performance dependence on wind speed, or on the type or absence of windscreens.

To test the system's ability to detect blasts within wind noise, a computer-based audio mixing system was used to create audio signals that included wind and blast noise. A digital audio mixing program allowed a blast waveform to be mixed in at any point in a wind noise signal, and at any relative level. These audio tracks were then played into the discriminator. Both the normal and low frequency blast waveforms were inserted into the wind noise files. The blasts were inserted at various relative levels and at various positions within the files. The detection rates observed during this testing were at least as good as those found in the Monte Carlo simulations mentioned previously. Unfortunately, no practical way was found to execute a random test in this situation. All that can be said is that the detection rates given previously seem to be good representations of the actual real-time performance.

Recordings of wind noise with blast noise present were made over 2 days near Fort Riley, KS. The blast noise source was from a training range approximately

4 km away where 120 mm artillery were being fired. Unfortunately, the normally high wind speeds seen in that area were not present during the 2 days of recording. The wind speed varied from 0 to 5 m per second (mps) with the average being close to 3 mps. The two microphones were spaced vertically with a 1.5-m separation, and the top microphone was 5.5 m above the ground. Standard 8-cm diameter foam windscreens were used. The average peak sound pressure level for the blast noise was 108 dB. The recordings were analyzed in the same manner as the recorded data mentioned earlier. A threshold of 100 dB was used for the analysis. All 62 blast events that occurred during the 7 hr of recording were detected by the discriminator. The discriminator also reduced the number of false events — due to wind noise from 239 events to 9 events — a 96.2 percent reduction.

Table 3. Blast detection algorithm tests on Fort Riley data.

Tape ID	Threshold sound pressure level (dB)	Threshold Events	Threshold Events with Discriminator	False Event Reduction (%)	Average Wind Speed (m/s)	Wind Screen Used	Total Recording Time (min)
1	115	724	19	97.4	9.4	None	37
1	120	572	10	98.3	9.4	None	37
1	124	405	1	99.8	9.4	None	37
2	95	359	10	97.2	4	8 cm foam	120
3	95	290	7	97.6	4	8 cm foam	45
4	100	219	8	96.3	3.6	None	4
5	100	509	7	98.6	3.6	None	7
6	105	335	11	96.7	5.8	8 cm foam	25
7	105	233	11	95.3	5.8	8 cm foam	38
7	100	176	5	97.2	5.8	Specialized*	38
8	108	500	12	97.6	8	8 cm foam	82
8	103	349	6	98.2	8	Specialized*	82

Results of Cross-Multiplication in Blast Noise Measurement Simulations

As a requirement of the blast detection algorithm, the new monitor uses two vertically spaced microphones. The effect of using the cross-multiplication of these two channels instead of the square of one channel to measure the peak and SEL of blast events was studied with computer simulations. It was thought that the accuracy of blast noise measurements taken in the presence of wind noise could be improved by cross-multiplication, which would cancel some of the wind noise due to its being relatively uncorrelated between the two microphones. Monte Carlo type simulations were developed in which a blast was randomly inserted into wind noise at a given relative level and measured on each channel separately using the cross-multiplied signal. These measurements were then

compared with the measurement of the blast without any wind noise. The results show definite improvement in the accuracy of measurements of both the peak and SEL of the blast when the cross-multiplied signal is used. Table 4 summarizes the results of more than 70,000 simulations at each insertion level. Note that the simulations were also repeated for the case of a 20-degree propagation tilt so that the blast signals were not exactly aligned at each microphone. As seen in Table 4, the results with the tilt were not significantly different than those without the tilt.

Table 4. Measurement improvement for cross product.

Blast level relative to the wind (Leq basis, dB)	Average peak level measurement improvement (dB)	Average SEL measurement improvement (dB)	Propagation Tilt
+ 5	2.5	8.8	None
+10	1.4	7.9	None
+20	0.2	3.4	None
+ 5	2.5	8.7	20º
+10	1.3	7.2	20º
+20	0.2	2.7	20º

5 Implementation Details

CERL Noise Monitoring and Warning System 98 software consists of two major types of software. The first type is the DSP code that runs on the TI TMS 32C040 DSPs. This consists of two programs for each of two DSP chips. One DSP chip accepts incoming data from the ADCs attached to the DSP board. It filters the incoming data (the C-weight filter is implemented as a second-order infinite impulse response [IIR] filter), records integrals and peaks, and passes data off to the second DSP chip for blast detection. The second DSP chip accepts decimated data from the first chip (it accepts one sample for every 24 taken by the A/D), and analyzes the data using a running Fast Fourier Transform (FFT) and cross-correlation tests. The cross-correlation and the relative strength of two FFT frequency bins versus the entire signal are then used to determine whether it is likely that a blast is in progress. All data are transferred to the host CPU (an Intel 486-based embedded PC card attached to the ISA backplane) through a shared memory buffer on the DSP card.

The second type of software is for noise monitor control. Running on the i486 host CPU, this program is responsible for control of the monitor. It records the data collected and analyzed by the DSP chips, identifies data likely to represent loud blasts, queues it, and sends it off to the base station. It also performs system maintenance tasks such as automatic calibration, environmental control, and resetting and configuring the DSP CPUs and other attached hardware. This software, along with MS-DOS, KERMIT (for remote management and code updates), some other utilities, and the DSP code, is stored in 1 MB FLASH ROM on the virtual disk board. Data that include DSP and host control code updates and the storage queue are stored in 1 MB of SRAM also on the virtual disk board.

DSP Software Details

Introduction

The new CERL noise monitor is functionally comprised of two main parts: the real-time DSP code and the PC interface program. The DSP code runs on two TI TMS320C40 DSP processors sited on a Signal Processing DPC40B board from Spectrum (Burnabary, BC, Canada). One of the two processors runs a blast recognition algorithm and the other handles real-time measurement processes

including frequency weighting, computing the sum-of-squares and maximum values, and providing all relevant information to the PC interface program. The PC interface program takes tenth-second sum-of-squares and maximum levels information from the DSP and computes the standard monitor metrics as well as handling communications and interpreting commands from the base computer.

This chapter describes the code written for the two DSP chips used in the new monitor. The two processors are identical, except that only the first processor (CPU_A) can access the A/D samples and communicate with the PC via the DPRAM. Communication between CPU_A and the other processor (CPU_B) occurs via several high speed buffered communication ports. The blast recognition algorithm running on CPU_B will be described first.

Code for CPU_B: Blast Detection and Wind Noise Rejection Algorithm

The algorithm implemented in this DSP program was designed in 1995 and is explained in more detail in Benson (1996). The program is named MON_2. Throughout this section, portions of the code will be referenced by line number from the listing of the code in Appendices D1 and D2 of this report.

The algorithm takes samples at a rate of 2 kHz and computes two frequency bins from a length 128 FFT. These bins are combined through squaring the individual points, cross multiplying across the two channels, and summing the cross multiplied values. This “energy” value is then stored in a circular buffer. The samples for each data channel are also stored in circular buffers. The normalized zero-lag cross correlation between the two data channels is also computed using a data block of length 64 (23 ms). The history of energy values is searched for its maximum value from the end of the record to the current time minus 70 samples. The current energy value is then divided by the maximum found and becomes the decision ratio. The algorithm decides that a blast is present if the decision ratio is above 25 and the correlation is above 70 percent. The program receives its data samples over C40 communications ports 3 and 4 from the program MON_1, which runs concurrently on CPU_A. MON_2 sends a blast detection flag to MON_1 via communication port 3. The integer number flag is equal to 1 if there is a blast and 0 if otherwise.

In lines 14-26, three circular buffers are initialized in memory. The section labels “chahist,” “chbhist,” and “ehist” are mapped to specific points in the processor’s memory map by the MON_2.CMD file (linker command file). The labels “CHA,” “CHB,” and “EH” point to the first position in each buffer that is initialized to hold all zeros by the .space directive. The data memory section begins at line 29. All sections are defined in the linker command file. Lines 31-42 define

the memory locations for the stack, the interrupt vector table, the data histories, and the control and buffers for the communication ports. Lines 44-85 simply make global variables of all the labels used in the program so that they can be used by name in the debugger. The rest of the data section (lines 87-131) serves to initialize several variables and to set up constants used in calculations. Note the value of the internal interrupt enable mask (IIEMASK), which will be used to "turn on" the interrupts associated with communication ports 3 and 4 from which the new data samples originate.

Line 134 starts the .text section, which contains the main code. The unconditional branch statement brings the program counter down to the start of the executable code. The .word directives on lines 137-155 define the contents of the interrupt vector table. Only two interrupts are defined: ICRDY3, which interrupts when a new word has come to the input buffer of communication port 3, and ICRDY4, which is similarly defined for port 4.

Line 158 marks the beginning of the executable part of the program. The data pointer, stack pointer, and interrupt vector table are loaded. Line 169 begins a section of code designed to set up the communication ports and empty out their input buffers. Lines 169-174 load the address of the ports and their control registers into address registers. Lines 176-181 halt the input FIFOs* for the two ports so that no new data comes in while the initialization process continues. Lines 184-194 test to see how many words are in the input FIFO for port 3. If words are present, they are read out to empty the FIFO. This same procedure is then done for port 4 in lines 197-207. Lines 210-211 enable the two interrupts ICRDY3 and ICRDY4 in the internal interrupt enable register (IIE) and then globally enable interrupts in the status register (ST). Lines 213-218 then restart the input FIFOs for the two ports. The last three instructions (220-222) before the main program loop simply load the circular buffer start points into address registers.

The main loop of the program starts on line 230. Lines 229-235 serve to test two flags, READYA and READYB, which are set high in the interrupt service routines that receive the data over the communication ports. The service routines (637-640 for channel A, port 3, and 646-649 for channel B, port 4) simply read from the input FIFO, store the data in a variable, and then set the READYA or

* FIFO – first in, first out refers to input queues.

READYB flag to 1. When both ready flags are equal to 1, the computation begins after the flags are reset to 0.

The main computation part of the program begins on line 244. First, the new data samples are loaded, converted to floating point, scaled down by 10-4, rounded, and written to the circular buffers. Before writing the current samples to the circular buffers, the oldest value in a length 64 sub-block of the buffer is read (lines 265-272) for use in the correlation computations. Line 295 starts the computation of the first frequency bin for channel A. The frequency bins (points 2 and 3 of a length 128 FFT) are computed using a continuous FFT (see Benson 1996 for details). These FFT point calculations occupy lines 295-396. Next, the square of the magnitudes of the two FFT points from each channel are computed (lines 404-429). Lines 436-443 compute the current "energy" value. Then the maximum energy value in the circular buffer (of length 690) is found in lines 449-458. The buffer is simply stepped through, comparing the value at that location to the maximum found so far during the search. Only the region from oldest to newest -70 samples is searched. Once the maximum value is found, its inverse is computed using the fpinv subroutine (lines 586-628) and the decision ratio is formed by multiplying this inverse by the current energy value (lines 459-464). The gap in the search region exists to ensure that a peak in the energy values (due to the presence of blast) does not get divided by a value within that peak area.

The next section of code computes the normalized cross correlation over the last 64 samples. Lines 473-487 compute the autocorrelation of channel A. For the purpose of this program, the autocorrelation is defined as the sum of the squares of the 64 most recent samples. This sum is computed by adding in the square of the current sample and subtracting the square of the oldest sample. This oldest sample value was found earlier in the program just before the new data were stored in their circular buffers. Key to the long-term stability of this calculation is the inclusion of rounding operations. If these are removed, small errors build up over time and create completely inaccurate correlation values. The autocorrelation for channel B is computed in lines 491-505. The cross correlation of channels A and B is computed in lines 510-537. First the sum of the cross multiplication of the last 64 samples is computed in the same manner as for the autocorrelations. Then, to get the square of the normalized zero-lag correlation ratio, the cross correlation value is divided by the product of the autocorrelation values. The square of the ratio is computed instead of the actual value because of the expense in computing the square root of a number.

The decision section of the program follows next. First, the value of the decision ratio is compared to the threshold level of 25 (lines 549-554). If the threshold is

not met or exceeded, a 0 is written to communication port 3 (574-576), signifying that no blast event is currently occurring. If the decision ratio threshold was met or exceeded, then the normalized zeroth lag cross-correlation value is compared to its threshold of 70 percent (558-563). If the correlation is greater than the threshold, then a 1 is written to communication port 3 to alert processor A that a blast event is occurring (576-570). If the correlation threshold is not met, then a zero is written to port 3. At this point, the algorithm is finished with the current sample's calculations, and the program loops back to line 229 and waits for the next pair of samples to become available.

Code for CPU_A: Noise Monitor Functions

The DSP code for CPU_A (MON_1) computes the raw data in tenth-second blocks (TSBs) needed by the PC interface program, which then computes the noise metrics such as peak SPL and SEL. The DSP knows nothing of the calibration settings or whether the unit is in a manual sample mode or in threshold mode. The raw data is passed from CPU_A to the PC interface program via the DPRAM.

With the inclusion of the blast recognition algorithm, this new monitor has two microphones. The two channels of data are sampled using a Loughborough Sound Images Crystal analog daughter module attached to the DSP carrier board. The A/D is a 16-bit dual-channel delta-sigma type. The module is programmed in the initialization section of MON_1 to sample at a rate of 48 kHz. MON_1 is also responsible for sending sampled data to MON_2 (blast recognition algorithm) at a rate of 2 kHz, requiring decimation by a factor of 24.

MON_1 computes the sum of squares of each channel in TSBs, the maximum value in the block, the position of the max within the block, and the number of blast recognition flags reported within the block. These computations are also done for the c-filtered version of each channel, the cross-multiplication of the two channels, and the c-filtered cross-multiplication. Since the blast recognition algorithm operates at a sampling rate of 2 kHz, blast recognition flags are only available every 24 samples. Accordingly, the TSBs are broken down into microblocks of length 24 samples. There are then 200 microblocks in every TSB. For each microblock, the sum of squares and maximum value information is computed and a blast recognition flag is gathered from CPU_B. The sum of squares information is then added up to create the TSBs and the maximum value is kept up to date through all of the microblocks to give a final maximum value for the TSB. The position of the maximum value is given only as the number of the microblock in which the maximum value occurred. A running total of the number of microblocks in which the blast recognition flag was high is also kept for the TSB.

The remainder of this section is a detailed, line-by-line description of the CPU_A code, MON_1. The program listing begins by setting up some data needed for the c-filtering operation. Line 32 sets the section to "data." Lines 34-43 enter the coefficients for the c-filter. Lines 51-69 create memory locations for the delays used in the c-filter. In lines 77-92 several important addresses are set up, such as the interrupt vector table, the locations of channel A and channel B data, and the communication ports. In line 93 the starting address of the DPRAM is defined. Line 94 designates a location in the DPRAM where the starting address of the data for the current TSB in the DPRAM. Lines 105-119 create variables used for storing the results of microblock calculations. The next portion of code is placed into a different area of memory as defined on line 126 in the .sect directive. The data in this section are sent to the PC interface via the DPRAM. To send this data quickly, indirect addressing is used, which necessitates the data starting at an easily known point in memory. This is accomplished by simply starting a new "section." The starting address of this section is set to a variable in line 121.

The data that are sent to the PC interface program via the DPRAM are stored in a circular buffer in the DPRAM. The buffer is of length 950, which corresponds to 50 TSBs (19 words x 50 blocks = 950). The memory space for this circular buffer is set up starting with the .sect directive in line 159. The section "DP_circ" is at the beginning of the DPRAM. The .space directive in line 161 fills 950 consecutive memory locations with zero, initializing the buffer.

Line 163 starts the program text section. Lines 169-185 comprise the interrupt vector table. This table shows that only one interrupt is defined, ANALOG (IIOF1), which corresponds to the new data samples arriving from the A/D. Lines 187-269 set global variables.

Line 274 starts the actual executable code. In lines 274-278 the data pointer, stack pointer, and interrupt vector table are loaded with their proper addresses. The next group of code in lines 282-302 sets up the A/D board for proper operation. Its sampling rate is set, it is reset and calibrated, and the key value is loaded into the analog configuration register. Next, in lines 304-307 the memory locations of the channel A and B data, the interrupt acknowledge, and the interrupt mask register are loaded into address registers for later use. In lines 311-314, address registers are loaded with the addresses of the output FIFOs for the communication ports used to transmit sample data to CPU_B, of the input FIFO from CPU_B that carries the blast flag, and of the beginning of the DPRAM.

The next section of code empties in the input FIFO for communication port 0, which contains the blast flags sent by CPU_B. This action will synchronize the

two processors. The number of words in the FIFO are read from the port's control register. The FIFO is then read repeatedly until all words are read. The DSP is then ready to enable the analog interrupt and begin collecting data. The IIOF1 interrupt is enabled in line 337 and the global interrupt enabled on the following line.

The noninterrupt service routine part of the program is contained on one line (341). Here the program waits for the A/D interrupt so that the next set of samples can be processed. The rest of the program is the ANALOG interrupt service routine.

The interrupt service routine begins with the reading of the two sample values from the A/D. The integer values are stored for future use as outputs (lines 355-56). The integer words are left shifted into the highest 16 bits for computation (lines 357-58). In lines 360-366 the decimation counter is loaded from memory and compared to 23 (decimating by a factor of 24: samples 0:22 thrown out). If the counter has reached 23, then the current samples are sent to CPU_B for use in the blast recognition computations. If it is not time to send the data, then the counter is incremented and the program moves on to the filtering stage. In the next section of code (lines 368-390), the current sample data are sent over the communication ports to CPU_B. Before sending the data, however, the program checks to see if any words are in the output FIFO of port 0. If one or more words are in the buffer, then the data are not sent to allow the two processors to "get in sync." After either sending the data or allowing for synchronization, the program moves on to the c-filtering operation.

The next portion of the code filters the current sample data using an IIR c-weighting digital filter. The code implemented here was largely taken from the TI TMS320C4x User's Guide 2. See appendix 3 of the user guide for an explanation of the design of this filter. The filtering of both channels of data fills lines 408-534.

On lines 541-544 the decimation counter is checked against 0 to see if the current samples will complete a microblock. The counter is compared to zero because the microblocks bound on the sending of decimated data to CPU_B and the counter would have been zeroed earlier in the program if this was the case. If it is not the end of a microblock, then the code continues with the sum of squares computations for "NOT END OF MICRO_BLOCK."

Lines 553-588 constitute the computation of the squares for the case where the current samples do not end a microblock. The sum of squares are computed by simply adding the square of the current data (flat and c-weighted) to running

sums for the current microblock. The multiplication of the two samples (again flat and c-weighted) is added to running sums as well. In lines 592-624 the absolute value of the current data samples (flat and c-weighted) are compared to the corresponding maximum values found thus far in the microblock. Also, the absolute value of the cross-multiplied samples is compared to its previous maximum value. If the individual values are not new maxima, then the program goes on to the next comparison. If the current data reflect a new maximum, then the value is written to a memory location for future comparison. On line 624 the program jumps to the end of the interrupt service routine, as all the computations needed for this case are done.

If the current samples ended the current microblock, then the program would have jumped from line 544 to line 638, where the computations for the end of a microblock occur. The first thing done in this section is to read the blast recognition flag sent by CPU_B over communications port 0. First, on lines 638-645, a flag called FIRST_TIME is tested. If this flag equals 1, then it is the first time that the blast flag is being read since the code started. No blast flag will be available from CPU_B the first time, so the program loads a 0 into the blast flag and then jumps to line 649 where the current flag is stored. It is then added to a sum for the current TSB. If it is not the first time, the blast flag is read on line 646 from communication port 0 and then stored and summed.

Next, on lines 658-691 the sum of squares is computed in the same manner as before. Then on lines 698-730 the final maximum values are determined for the microblock using the same method as before. Now that the microblock calculations have been completed, they need to be added to the current TSB. On lines 738-761 the sum of squares value for each channel (flat and c-weighted) and the cross multiplication of channels are added to a corresponding value for the current TSB. Then in lines 769-806, the maximum values found for the ending microblock are compared to the maximum values for the TSB. If a current microblock maximum is larger, the new value is stored and the microblock number (0-199) is stored as the position of the maximum within the current TSB. In lines 813-825, the microblock variables are cleared so that they are ready for the next block.

Now, the microblock counter is checked to see if the current microblock ends a TSB. If it does not, the counter is updated and the program jumps to the end of the interrupt service routine. If the block does end a TSB, then the program goes on to line 847, where the TSB data are moved to the appropriate location in the DPRAM for the PC interface program to read it. First (on line 847) the size of the circular buffer in DPRAM is loaded into the block size register to allow for the circular addressing. The current value of AR7, the address in DPRAM where

the first word will be written, is loaded into the location in DPRAM called CURRENT_ADDR so that the PC interface program knows where the most recent data are being written. Note that AR7 has not been modified since the last time the buffer was written and the last write statement incremented AR7 so that it points to the correct position. In lines 855-905 all the tenth-second data variables are stored to the DPRAM circular buffer. Indirect addressing is used so that parallel loads and stores can be implemented. All of the floating point numbers among the data are first converted to IEEE* format, which is most easily read by the PC interface program. Lastly, the microblock counter and all the TSB variables are zeroed in lines 909-935.

The last section of code is the end of the interrupt service routine. In this section the current sample for channel A is output to channel A on the D/A converter, and the blast flag is shifted (multiplied to make larger) and output to channel B of the D/A converter. The program then returns to line 341, where it waits for the next samples to be available and the processes to begin again.

Controller Software Details

The host software forms the interface between the DSP code and the outside world. The host code controls all of the external hardware on the monitor: the microphone calibrators, the modem, fans, the heater, and the wind meter. (The microphones and the microphone amplifier are largely self-contained and simply feed directly into the DSP board.) The noise monitor host code was developed using Borland C++ version 3.1 on the MS-DOS platform. It uses low-level calls, input/output (I/O) and inline assembly. These dependencies make it unlikely that the code will compile on a more modern compiler.

Controller Software Run-time Files

Table 5 is a list of files contained in the FLASH ROM (A:) on the noise monitor and a description of each file's function.

* IEEE = Institute for Electrical and Electronics Engineers

Table 5. Files contained on the noise monitor's FLASH ROM.

Size	Filename	Description
412	CONFIG.SYS	DOS configuration file
54645	COMMAND.COM	DOS command processor file
533	MONITOR.BAT	Noise monitor and field service mode startup file
632	C4XLOAD.ROM	DSP loader software
2924	NETAPI.CFG	DSP board configuration
1103	BOARD000.CFG	DSP board configuration
906	BOOT.OUT	DSP boot software
1010	EDBOOT.OUT	DSP loader software
8588	MONDSP_1.OUT	Field unit DSP software for CPU #1 (C-weight filter)
6896	MONDSP_2.OUT	Field unit DSP software for CPU #2 (blast detection)
748	EDLOAD.ROM	DSP loader software
6908	CMD_HELP.HLP	Field unit command help file (HELP <command>)
105116	KERLITE.EXE	KERMIT executable for field service mode
1076	MSRL314.PCH	KERMIT patch file
4320	MSKERMIT.INI	KERMIT initialization file
1126	MSCUSTOM.INI	KERMIT initialization file
167116	MAIN.EXE	Field unit controller software executable
14019	OPT_HELP.HLP	Field unit parameter help file (HELP SET <option>)
22856	WDMTSR.EXE	Watchdog TSR for field service mode
221	AUTOEXEC.BAT	Boot batch file
710	MON_CFG.DAT	Default (ROM) configuration file
710	BAK_CFG.DAT	Copy of the default configuration file
24681	DOS/LIST.COM	File viewer utility
22974	DOS/FORMAT.COM	MS-DOS disk formatting utility
16930	DOS/XCOPY.EXE	MS-DOS file copy utility
15718	DOS/DEBUG.EXE	MS-DOS debugger
12241	DOS/CHKDSK.EXE	MS-DOS file system check utility
17164	DOS/SLED.COM	Simple text file editor
5406	DOS/DRIVER.SYS	MS-DOS disk driver (used for SRAM and FLASH disks)
29136	DOS/HIMEM.SYS	MS-DOS high memory manager
11917	DOS/PCFBERAS.EXE	Utility to erase PCF-1 FLASH ROM
5861	DOS/DOSKEY.COM	MS-DOS command-line editing utility
29378	DOS/PKUNZIP.EXE	Uncompresses ZIP-compressed files
18319	DOS/MOVE.EXE	MS-DOS file move utility
28729	DOS/SLED.DOC	Text editing utility
40001	DOS/PCFBCOPY.EXE	Utility to update PCF-1 FLASH ROM
10748	DOS/DISKCOMP.COM	Floppy-disk compare utility
13335	DOS/DISKCOPY.COM	Floppy-disk copy utility
5873	DOS/RAMDRIVE.SYS	MS-DOS RAMDISK creation driver
30	DOS/REFLASH1.BAT	Part of automatic reflash utility
5	DOS/REBOOT.COM	Reboot the system
2949	DOS/PCFSRDVR.SYS	PCF-1 SRAM disk driver
97	DOS/REFLASH.BAT	Automatic reflash utility (hit "3" after numlock light turns on)
61	DOS/KILLROM.BAT	Erase SRAM disk (hit "4" after numlock light comes on)

The 1 MB SRAM disk (C:) on the noise monitor is used to store the data queues and configuration information. Substantial extra space is also available on the SRAM disk for field upgrades — both the field unit control software and the DSP software can be upgraded by sending new code into the SRAM disk. The DATA directory in the SRAM disk contains two copies of the data queue (MON_DATA.DAT and BAK_DATA.DAT). The two copies are updated separately. If for some reason an update is interrupted, the corrupted queue file will be rewritten using the data from the remaining queue file. This gives the data queue substantial immunity to power failures. (In testing, data were occasionally lost before the wind meter IRQ bounce problems were corrected. Since then,

no queue failures have been observed.) These queues are 256 K (262,144 bytes) long and hold up to 2,048 noise events. A checksum is used to verify on-disk and in-memory data as a guard against corruption. If these files are not found or are found to be corrupted, an empty queue will automatically be created.

The monitor's configuration is stored on the SRAM disk in the 710-byte files "MON_CFG.DAT" and "BAK_CFG.DAT." These two files are updated independently, and both provide a checksum to guard against corruption. These files contain the calibration data, thresholds, and all of the "SET" option parameters. If these files are not found or are found to be corrupted, the default configuration will be copied from the MON_CFG.DAT file stored in the FLASH ROM.

Controller code updates are stored on the SRAM disk in the file "MAIN.EXE." If this file is present, the startup code will automatically execute it instead of the version of the controller code stored in ROM. Some rudimentary protections against invalid controller code exist, but it is entirely possible that the field unit will be rendered inoperable by an invalid version of MAIN.EXE stored in the SRAM. Therefore, it is very important to use test controller code updates extensively before updating the field units, and to exercise caution to prevent invalid or incomplete MAIN.EXE files from being stored in SRAM.

Controller Software Boot Process and Field Service Mode

The field unit controller boots into MS-DOS and uses the standard MS-DOS AUTOEXEC.BAT and CONFIG.SYS files to control the boot process. These files are responsible for setting up the field unit's environment, as well as handling the transition into field service mode and rewriting the FLASH ROM. All of the MS-DOS batch files and configuration files are attached as Appendix D.

The CONFIG.SYS file includes a standard boot menu, which can be used to perform various maintenance functions. The default option allows the field unit to boot normally and collect data. If another option is chosen (by attaching a keyboard and hitting a number plus <Enter> after the NumLock light comes on), the field unit will go into an administrative mode:

1 Normal running; default

2 Service mode; boot to an MS-DOS prompt. Requires a VGA card

3 Reflash mode; erase and reprogram boot FLASH ROM from a floppy disk

4 Reset all stored data; erases the queue, configuration, and program updates on the SRAM disk.

The CONFIG.SYS file also loads a few required device drivers. These drivers include PCFSDRVR.SYS, the driver for the SRAM virtual disk, and RAMDISK.SYS, which creates an in-memory virtual disk that is used as temporary storage or "scratch space" during the field upgrade procedure.

The AUTOEXEC.BAT file automatically sets up a few environment variables, such as the search path and the DSP code location, and then runs the correct code for the specified operation mode. If a reflash or a reset is specified, that code is dispatched; otherwise, the file "MONITOR.BAT" is called. This batch file checks to see if an update to MAIN.EXE is stored on the SRAM disk. If so, it executes the update. If not, it executes the version of MAIN.EXE stored in the FLASH ROM. If for some reason this first invocation of MAIN.EXE fails, the monitor will automatically attempt to start the version stored in the FLASH ROM.

The monitor startup batch file also handles the field unit's transition into service mode. (The field unit transitions out of service mode by rebooting.) This is done using the exit code returned by the field unit's control software to the batch file. A return code of 101 or 102 indicates that the field unit should transition to service mode — 101 means that the connection will be on COM1, and 102 means that the connection will be on COM2. The batch file then starts the watchdog TSR (WDMTSR.EXE), which reboots the system when the indicated COM port disconnects, and starts KERMIT. KERMIT automatically loads a configuration file written by the field unit control software on startup. This file indicates the COM port and speed to use for the link. The watchdog TSR monitors the communication link. If a link loss is detected, the TSR will allow the system to restart after waiting approximately 1 minute for any pending operations to complete. In this way, the field unit will automatically restart without corrupting any data if the remote user should fail to explicitly log out of field service mode.

Source Code Overview

The field unit's overall control software is not a true real-time system; it runs a cycle of various tasks to be performed, but no exact timing measurements of these tasks has been made. Generally, the 486DX2/66 control processor is fast enough to keep up with the tenth-second interval between blocks coming from the DSP board; however, this is not guaranteed. In fact, certain tasks (such as updating the queue data in the SRAM disk) can take longer than a tenth of a second. However, because file system accesses are short (each access only

updates one “page” of the on-disk memory queue or the configuration file) and spaced between 15 DSP checks, the 50-element circular queue on the DSP board will smooth out these delays. Therefore, in practice, no data from the DSP are lost. However, the queue contents are reread when a threshold block is triggered — the pre-trigger data are read out of the older TSB still in the queue. Therefore, at most 4.5 seconds of pre-trigger data is available; otherwise, delays may cause incorrect data to be read from the queue.

The field unit control software is split into several modules. Each of these modules consists of a C or C++ source file (.CPP or .C extension) and a header file that contains the declarations for functions and variables used in that module. The source modules are split up into the following functions.

- DSP.CPP: Handles all data collection and the interaction with the DSPs. Retrieves data from the shared memory, accumulates the tenth-second DSP blocks into longer blocks. Identifies data exceeding predefined threshold values to trigger data storage. Also accepts and accumulates data from explicit commands and microphone calibration. It also maintains a queue of commands that will require future attention.
- DSPERR.CPP: Prints out error messages on DSP initialization.
- PROTOCOL.CPP: Handles the protocol for communication with the base station. Responsible for receipt, generation, and acknowledgment of data blocks, placing and receiving calls, modem initialization and control, transmit pacing, and other low-level data communications functions.
- SERCON.CPP: Provides functions that allow the field unit’s physical console to be reflected to the modem or serial port.
- CMD.CPP: Contains the code that parses console commands entered by the user and the code for implementation of all of the console commands.
- MAIN.CPP: Contains all of the initialization and shutdown commands and the main loop. Also contains the code that handles the field unit console, and triggers certain periodic tasks such as automatic calibrations.
- DATA.CPP: Controls the maintenance of the data queue, both in memory and on disk. Responsible for synchronizing the main data queue with the virtual disk, maintaining redundant copies, as well as allowing FIFO access to data in the queue by the noise monitor.
- CONFIG.CPP: Maintains the noise monitor configuration structure (struct config_struct cfg); synchronizes the in-memory configuration data with the configuration on disk. Also contains the compiled-in defaults for bootstrapping.
- PACKET.CPP: Responsible for the high-level communications between the base station and the monitor. Handles the high-level block format, sending queue data, and base-station control. This module duplicates the block handling characteristics of the original CERL noise monitor. To do this, it drops

a substantial amount of data collected by the new software. It may be replaced to allow this data to be collected.

- DATACONV.CPP: Converts IEEE floating point, integral, and date values from noise monitor internal data into the ASCII* data formats expected by the base station program. Used extensively by the high-level packet construction in PACKET.CPP.
- PPORT.H: Contains code to control the parallel port data output, which is used to control the microphone calibrators, status LED, fan, and heater.
- WIND.CPP: Includes code to read the wind meter interrupt, keep track of the passage of time, and return the actual wind speed (in miles per hour). Also maintains a database of different wind meter types (currently only used for console display).
- ENVIRON.CPP: Tracks current environmental conditions (specifically, ambient temperature inside the noise monitor casing and voltage supplies), issues warnings when necessary, and attempts to keep the temperature within an acceptable range using the heater and fan.
- WDT501P.C: This module controls the Industrial Computer Source WDT-501P watchdog timer board. This board contains several components, including a thermal sensor, voltage sensors, the watchdog timer, and isolated inputs, all of which can generate an interrupt. This module initializes the board, starts the watchdog, and handles interrupts coming from the board. (These interrupts are passed to MAIN.CPP, which ignores most interrupts and dispatches the wind meter interrupts coming from the isolated inputs to WIND.CPP.)
- IBMCOM.C and CIRC_BUF.C: These two modules, put together, form the serial driver used for communication with both the serial console and modem interface.

At the highest level, the flow of control in the field unit control software is very simple. The MAIN module initializes the hardware and software modules in the field unit software, loads the configuration data and the data queues from the SRAM, and then calls the DSP module to synchronize the controller with the DSP. Then the field unit enters the main loop, which cycles through the following tasks:

- check for an incoming tenth-second data block from the DSPs (calls DSP.CPP)

* ASCII = American Standard Code for Information Interchange

- check DSP service event coming due (calls DSP.CPP)
- check for incoming base station activity (calls PROTOCOL.CPP)
- update the status displays
- perform closed-loop environmental control (calls ENVIRON.CPP)
- flush an unwritten queue or configuration data block to disk (DATA.CPP and CONFIG.CPP)
- schedule automatic calibrations if due
- restart serial port drivers if necessary (so the baud rate can be changed on the fly)
- check for midnight and schedule a reboot if so.

The main loop is executed until a shutdown is requested (by setting the quit variable to 1). After a shutdown is scheduled, the main loop exits, and the system shuts down. (Going into service mode also shuts down the field unit; it simply sets the return code variable to 101 or 102 to indicate that the field unit is going into service mode.) First, any unwritten queue and configuration data are written out; then the hardware is shut down. The hardware watchdog is then programmed to reboot the system in 30 seconds. This step was added to prevent a failure starting up KERMIT or the watchdog TSR from causing the field unit to become unresponsive.

Data Queue File Format

The data is stored on disk in two identical queue files (MON_DATA.DAT and BAK_DATA.DAT). Both on disk and in memory the queue is stored as a circular buffer. Note that the head and tail pointers are not stored on disk, so all of the empty blocks in the queue are required to be contiguous. The tail (oldest element) of the queue is located after the last empty block, and the newest element in the queue is located before the first empty block. (Empty blocks are flagged by 0 (NUL) in the type field.) At least one empty element must always be in the queue to ensure that the top and end of queue pointers can be set when the data are loaded.

All queue elements are 128 bytes. If the DataBlock structure (see below) is changed, update the PAD field to ensure that the queue element remains exactly 128 bytes. The queue is updated on disk in 4-K pages (32 queue elements). The checksum should be checked whenever a queue element is used to guard against data corruption. Note that an easy way to delete a data block is to set it to all zeros; the correct checksum for an all-zero data block is zero.

struct DataBlock

{

char type; /* A Threshold block

- B Calibration block
- H Fixed length manual block
- M Arbitrary manual block
- E Temperature warning block
- F Voltage warning block

* *\0 Empty block*

*/

```
   char     status;    // Status flags from WDT501P

   float sum[4];       // 0 – flat, channel 1       1 – flat, channel 2
   float peak[4];      // 2 – C weight, channel 1   3 – C weight, channel 2
```

unsigned long int peakPos[4];

unsigned long int BDAblips;

```
float cross_sum[2];   // 0 – channel 1
```

float cross_peak[2]; // 1 – channel 2

unsigned long int cross_peakPos[2];

unsigned long int sample_length; // in tenth-second units

char windy;

```
   signed char temp;    // Temperature, from WDT501P

   enum Thresholds threshold;    // what threshold flag was exceeded

   struct time btime;   // DOS time block was taken
   struct date bdate;   // DOS date block was taken

   char flag;          // Flag a data block as important (warrants a call)
   char filter;        // Flag that the data block passed the filter

   char pad[30];       // Pad to 128 bytes.

   unsigned checksum;   // 16-bit checksum.
                        // Start with 0.
                        // For each 16-bit word in the block,
                        // add that word in and rotate checksum left 1 bit.
                        // Checksum value is this number minus 65536,
                        // such that the checksum of a correctly encoded
                        // entire block is 0.
};
```

Configuration Notes

The monitor includes some configuration options that allow the behavior of the field unit to be changed. These options are preset in the boot image and are intended to allow the controller code to be adapted to different external hardware.

Configuration and data redirection

The first option is the REDIRECT command, which allows for the configuration or the data queue to be located somewhere other than the initial directory from which the controller code was started. This facility is used in the FLASH ROM image to allow configuration defaults to be stored in the ROM (which give the monitor enough information about its hardware for the base station to contact it and configure it).

Typing "REDIRECT CONFIG <path>" tells the monitor to read its configuration from a configuration file stored in the directory indicated by <path> instead. This request is written in the configuration file from which the monitor last read its configuration — in other words, if the field unit was redirected from the current directory to the new location, the redirection request will be written in the new location, not the current directory. When the controller code is restarted, it first reads the configuration files from the current directory, then, if a redirect is indicated, it reads the configuration file from the designated location. Note that an indefinite sequence of redirects is permissible. If the redirection points at a directory in which no configuration files exist, a configuration file will be written in that directory; the defaults will be taken from the configuration file that pointed at that directory.

This facility is used in the monitor by storing a redirection to C:\ (the SRAM virtual disk) in the configuration file in ROM. This allows the hardware defaults (such as the field unit number) to be stored in the FLASH ROM configuration, while still allowing configuration changes to be made on the fly.

"REDIRECT DATA <path>" directs the field unit to find its data queues in an indicated directory. Note that there is no chaining for the data queues. If no data queue exists in the indicated directory, an empty queue will be created.

"REDIRECT LOCK" locks out further redirections. Note that REDIRECT LOCK is set in the field unit's FLASH ROM, so these commands do not actually work in the field unit. (These parameters need not be changed after the field unit is initially set up, as all the relevant data is stored in the unit's SRAM virtual disk.)

B&K 4184 mode

There is some support for the B&K 4184 microphone (Bruël & Kjær, Denmark) in the field unit code. Unlike the Larson-Davis microphones, which have an electrostatic calibrator, these microphones have an acoustic calibrator. These acoustic calibrators have two important limitations: they do not have a consistent level (so they are useless for calibration), and they generate noise that can be picked up by the other microphone on the field unit (so they cannot be used at the same time). For this reason, a 4184 mode is included that calibrates the two microphones one at a time, and does not update the internal microphone calibration coefficients.

SET 4184 ON enables the staggered calibration and disables updating the calibration coefficients from the calibration data.

SET 4184 CAL staggers the calibration. (Note that the settle times allowed for the staggered calibration mode are shorter, so the total amount of time taken for the calibration remains roughly constant.)

SET 4184 OFF disables the support for the B&K 4184 microphones. This should be used for the Larson-Davis microphones on the prototype field unit.

Wind meter setup

To allow the current wind speed to be displayed when the unit is in console mode, it must be able to convert the wind meter's output (in pulses per second) into miles per hour. This is done by a pair of parameters (addend and multiplicand) using the following formula:

$$\frac{Pulses}{\text{sec}} = ((multiplier \cdot windspeed) - addend)$$

These parameters can be set using an internal table (WIND.TXT in the FLASH ROM), which contains mappings from symbolic names of wind meters (such as W203 for a W203 wind meter, or W203PC for a W203 wind meter with plastic cups). To set these parameters, type "SET WINDMETER <name>" where <name> is the symbolic name for the wind meter. To set the addend and multiplicand directly, type "SET WINDMETER <multiplicand> <addend>."

Channel mode

Since the field unit has two microphones, it is sometimes useful to collect data from only one of them. (Note that the blast-detection algorithm always uses both microphones.) If one microphone is significantly out of calibration or taking bad data, it can be ignored. To do this:

SET CHANNEL_MODE CH1	Only take data from the top microphone (Channel 1)
SET CHANNEL_MODE CH2	Only take data from the bottom microphone (Channel 2)
SET CHANNEL_MODE MEAN	Return data from both microphones.

If CHANNEL_MODE is set to MEAN, the field unit returns the geometric mean of the two microphone peaks and the cross-correlation between the two microphones. If the CHANNEL_MODE is set to CH1 or CH2, the peak and sum-of-squares values for that microphone are used. Note that setting CHANNEL_MODE to MEAN will tend to reject wind noise, which is less correlated between the two microphones than blast noise.

Known errors

A minor programming error prevents the absolute threshold from working properly in the code stored in the field unit FLASH ROM at Fort Drum. The latest version of the controller code has this problem corrected. The code is believed to be Year 2000 compliant, but extensive testing has not been done.

Hardware Details

The external enclosure has been modified internally to accept the new hardware. Modifications include additional mount points for the card cage on the right and an electrical box for the heater installed on the left. The modem cage on the top right was removed and replaced with a strap to hold the Larson-Davis microphone amplifier and power supply. The mounting for the 24-V board and the 24-V transformer were also moved somewhat to provide additional space for the ISA card cage.

The main electrical subsystems of the redesigned monitor include:

- AC power conditioning and distribution, including serial and parallel line protectors, a power switch and an electrical box, a 3-A fuse for the protected

power supplies, a 10-A fuse connected to the heater power supply, and a line protector for the incoming modem line.

- An ISA card cage, containing all of the processing and communication hardware. (The card cage replaces all of the functionality of the CIM card cage in the original monitor design, and also includes an internal modem.) The card cage includes:
  - an Industrial Computer Source 486-based CPU card
  - an Industrial Computer Source Watchdog Timer / Environmental Monitor board
  - Spectrum Signal Processing Dual-TMS32C040 Digital Signal Processor Board
  - Curtis FLASH ROM/SRAM disk emulator card, for program and data storage
  - USR 33.6kbps internal modem for communication.
- ISA power supply unit (also used to provide +5 V and +12 V to other devices)
- A Larson-Davis outdoor microphone system. (The Larson-Davis microphones have been problematic. It is recommended that a different brand be used in any future monitors based on our design.) The audio outputs from the microphones attach to ADCs installed on the DSP board in the card cage. This microphone system includes:
  - two Larson-Davis 2100 Outdoor Microphones
  - Larson-Davis Microphone Dual Amplifier/Power Supply
  - CERL-made cable harness and mounting boxes
  - two CERL-made audio input and control boards, and an audio input assembly.
- A CERL noise monitor 24-V board and transformer assembly. This is identical to the original Noise Monitor's 24-V board, except for an added LED to provide operational feedback.
- Environmental control subsystem: 1000-W, 120-V AC plug-in heater and a 24-V DC fan, both controlled by the 24-V board. These provide a closed-loop environmental control inside the monitor enclosure.
- Qualimetrics W203 wind meter unit
- Hardware debouncer unit for wind meter input. This board (added to solve problems found in testing at Fort Drum) prevents slow wind speeds from crashing the unit by taking the input from the wind meter and generating a clean square-wave output to send into the watchdog board's digital input.

Configuring ISA Cards

DSP board

The dual Spectrum C40 DSP board system consists of several components, and must be ordered as shown:

600-02057 DPC40-40MHz Dual C40 System Board

100-02057 DPC40 Document Kit

202-02058 Cable Kit

600-02058 MDC40S1-50MHz C40 with 384KB SRAM (Quantity 2)

100-02048 MDC40 Document Set

600-02050 AM/D16DS: Dual 16-bit 50KHz Delta Sigma Analog Converters

100-00250 AM/D16DS Document Set

The DSP board must be configured after arrival. The board will arrive pre-assembled, so it is not necessary to install the modules. However, the jumpers must be properly configured, and the boot ROM for the DSP microprocessors must be written.

To do this, the jumpers must first be configured. The configurations for the two C40 modules are:

Left Module:	LK1:	1 = *	(A)
	LK2:	1 * =	(B)
	LK3:	1 * *	
	LK4:	1 =	
Right Module:	LK1:	1 = *	
	LK2:	1 * =	
	LK3:	1 =	

LK4: 1 =

Carrier board: LK8: = * LK1: = *

(left to right, this looks like = * = *)

LK6: =

=

* *

* *

=

=

=

Second, the DSP's parallel ports must be connected. Connect them by attaching the included port cables between headers J14 and J7 (primary module port 0 to secondary module port 3) and J4 to J6 (primary module port 1 to secondary module port 4).

Third, heatsinks must be installed on the DSP microprocessors by using thermal glue and a 486-style heatsink with one of the "lips" on the side filed off. The heatsinks are important to prevent problems in the harsh conditions the monitor can be subject to.

Last, the boot ROMs on the DSP modules must be programmed. To do this, the DSP board must be inserted into an ISA bus slot in an IBM PC-compatible machine. Program by inserting the Monitor PEROM* programming disk, and doing the following:

1. Run the "PEROM" program.
2. Press "C" and hit <return> (to continue)

* PEROM = programmable, erasable, read-only memory

3. Press “A” and hit <return> (to program all processors in the network)
4. Choose “A” (the DPC/C40B)
5. Choose “Y” (to make IDROM bootable).

At this point, the PEROM program should say that both modules were “blown OK” and that programming was completed.

Figure E1 shows the audio input connection on the assembled DSP board. This connection is closest to the ISA bus edge connector, and carries the analog signal from the microphone preamp into the ADCs on the DSP board.

PCF-1 Disk Emulator

The PCF-1 Disk Emulator (Industrial Computer Source, city) part numbers are:

PCF1A-0MFS Flash/SRAM Disk Emulator

PCF1-FE11MB Flash EPROM SIMM

PCF1-SR11MB SRAM SIMM

To configure the PCF-1 Disk Emulator board, first install the two single inline memory modules (SIMMs) and the lithium backup battery. The first slot (the lowest slot on the board) must contain the FLASH SIMM; the second slot (above it) contains the SRAM SIMM.

All of the onboard jumpers are left at their factory settings (no jumpers). The Dual Inline Package (DIP) switches on the back, however, must be set. First set both the DIP switches on the back of the emulator board to ON (programming mode.) Then insert the PCF-1 Support Disk, and run the setup program PCFSETUP.

Use the following responses to PCFSETUP’s prompts:

Floppy or Hard Drive Emulation (F/H)? [F]	Type: F
Single or Dual disk emulation (S/D)? [D]	Type: D
SIMM Type in socket P1, 1M Flash or 1M SRAM (1F/1S)? [1F}	Type: 1F
Boot Disk Size (A=3.5” HD, B=5.25” HD)? [B]	Type: A

SIMM Type in sockets P2-P7, ... (1F/2F/1S)? [1F] Type: 1S

Total number of SIMMs in sockets P2-P7 (1-6)? [4] Type: 1

Boot Memory Address (A=CA00, B=CC00, C=CE00, D=D000)? [A] Type: A

DMA channel (0-3)? [1] Type: 1

Base I/O Port Address (B=2B0, C=2C0, D=2D0, E=2E0)? [B] Type: B

This procedure should be acknowledged by the message “done” on the monitor.

Next, the boot FLASH SIMM must be programmed by removing the PCF-1 Support Disk and inserting the image disk. Then run the program “PCFBCOPY” by:

Switching to the directory “A:\DOS” on the master disk.
Running the program “PCFBCOPY A: F”

PCFBCOPY will automatically erase the FLASH SIMM and write the monitor's program from the image disk into the FLASH on the disk emulator board. The programming computer should then be turned off, and the PCF-1 board removed. Finally, the PCF-1's configuration switch SW1-2 must be turned OFF. (SW1-1 is left ON.) This allows the boot SIMM to replace floppy drive A: when the board is installed in the field unit. When this is done, the PCF-1 is ready for installation.

Watchdog timer board

The watchdog timer board (with options) is also from Industrial Computer Source. The part number is:

WDT-501P Watchdog Timer with isolated input and voltage and temperature monitors.

The watchdog timer board is configured by setting a bank of DIP switches to set the base address. These switches must be set to achieve the base address of 0x2A0 expected by the monitor software. This is done with the following combination of switch settings:

A9	OFF	1
A8	ON	0
A7	OFF	1
A6	ON	0

A5	OFF	1
A4	ON	0
A3	ON	0

In addition, a jumper must be placed on the IRQ 15 jumper.

Also, a wire must be soldered into a Berg-type shunt connector and connected to the ~WDRST line (the bottom-most screw lug on the left side of the board).

Internal modem

The modem must be jumpered for COM1, IRQ 4. Any ISA internal modem that is not "Plug and Play" will work well. The prototype field unit used a USR Sportster 33.6 kbps modem. Follow the directions in the modem manual to set the IRQ and COM port for the modem.

CPU board

The CPU board and SIMMs (Industrial Computer Source) part numbers are:

SB4862PVN/66	Single Board Computer: 486DX2/66 CPU, 128 K Cache, No Video
TMS51236-60	2 MB Fast Page Mode Parity 72-pin SIMMs (two required)

The CPU board must have two 2 MB or 4 MB SIMMs installed to achieve 4MB or 8 MB of memory. (The monitor only requires 4 MB, but 2 MB SIMMs are difficult to find.) The required SIMMs are standard 70 ns, parity, fast-page-mode 72-pin (512Kx36 or 1Mx36) SIMMs. These SIMMs are installed in the slots at the very left edge of the board.

Video board

Any VGA video card can be used for system setup. No configuration is necessary for this board.

Assembling the ISA Card Cage

It is best to assemble the ISA card cage "on the bench" before installing it in the monitor. Quite a bit of configuration must be done on the assembled card cage

before it will work in the monitor itself. The card cage and power supply (Industrial Computer Source) part numbers are:

OEMC06	6 slot OEM ISA Card Cage
OEMC-P25	250 watt AT Power Supply

First, attach the power supply to the card cage. The connectors for the power supply are installed with the black wires in the center of the two connectors. The cards should be installed in the card cage in this order (from left to right, with the ISA bus connectors at the top of the cage):

Slot 1:	CPU Board
Slot 2:	Watchdog Board
Slot 3:	VGA Card
Slot 4:	PCF-1 Virtual Disk board
Slot 5:	DSP board
Slot 6:	Internal Modem

Then, attach a 9-pin serial connector to the cutout in the upper-left corner (by the CPU board), and attach it to the “Serial Port 2” connector on the CPU board. (There are two 10-pin serial connectors on the CPU board. Serial Port 2 is the one farthest from the 25-pin parallel connector.) Attach the Berg shunt from the watchdog board to pin 1 of the “RESET” connector, and attach a keyboard to the 5-pin keyboard connector. Finally, attach a monitor to the VGA card. Connections to the ISA backplane are shown in Figure E2. Table E1 lists the pin assignments on the connector on the back of the watchdog board.

To configure the CPU board’s Complimentary Metal Oxide Semiconductor (CMOS) static RAM (SRAM), one needs to enter its internal setup program. To do this, hit <DEL> on the keyboard as soon as the BIOS banner displays. The following settings need to be made in the BIOS:

Under Standard Setup:

- Set day, date, and time.
- Set Floppy A: to 1.44 MB 3.5 in.
- Set Floppy B: to Not Installed Under Advanced Setup:
- Set System Keyboard to Not Present
- Set Wait For “F1” If Any Error to Disabled Under Peripheral Setup:
- Set Serial Port 1 to Disabled
- Set Serial Port 2 to 2F8
- Set Parallel Port to 378

When this is complete, disconnect the keyboard and reboot the card cage. The noise monitor should start up. DSP startup should indicate no errors, and the main part of the display should indicate "STARTUP COMPLETE." It will also indicate that the data queues are not found and were being rewritten (this is normal). The last step of setup is to set the unit serial number. To do this, type the command "SET UNIT <number>" followed by "SET UNIT LOCK" to prevent the unit number from accidental changes.

At this point, the monitor card cage is fully functional. The last part of assembling the card cage is removing the VGA board. It is important that this board be removed and the hole in the backplane not be covered so the ISA boards are properly cooled. It is also recommended that the slot covers for the other cards have holes drilled in them to allow for some airflow. In addition, a 486-chip fan was mounted under the card cage by the DSP chips in the prototype unit to provide additional airflow across the DSPs.

Assembling Custom Components

The CERL Noise Monitoring and Warning System 98 field units use a number of custom components that must be assembled. These components include:

- two heavily modified audio input printed-circuit boards (PCBs)
- audio input module
- modified 24-V board
- control wiring harness
- microphone input harness
- debouncer circuit
- audio cables
- microphone enclosures
- main electrical box
- heater electrical box.

These devices must be assembled to match the described specifications.

Audio input PCBs

The audio input board for the original field units was designed to support a single B&K 4921 microphone (which has an onboard bias power supply) and to provide a mechanism enabling the field unit's microphone to be attached to the telephone line. For the redesigned field unit, the same PCB was used for an entirely different purpose. The PCB was reused only for convenience — many of its

traces were cut or left unused, and several jumper wires were attached. For the new field unit, the audio input boards served two purposes:

1. To protect the monitor's internal circuits by providing surge suppression on the wind and audio signal lines. (A design flaw in the audio input PCBs mounted in the prototype disables the surge suppression on the wind meter signals. This is corrected in the latest schematics.)
2. They provide the circuits to take the TTL-level calibrator controls from the card cage and provide the current loop used to control the calibrator on the Larson-Davis microphones.

The primary use of the audio input PCBs, however, is to physically accept the connector used to attach the two microphones and the wind meter. This connector is a female 12-pin Amphenol round connector. The following pins are used to attach the Larson-Davis microphone and wind meter to the prototype unit:

A Ground
B Shield Ground
C Audio Signal
D Calibration Control
E Bias Voltage
F Ground
H Wind meter (+)
J +9 volts
K -9 volts
L Wind meter (-)
M Ground
N +12 volts

The audio input boards should be built using internal connections matching the "Modified Audio Board" design shown in Appendix E. Externally, the audio boards both have a Molex connector (which brings in the ground, power, and bias for the outdoor microphone units) and a 1/8-in. headphone-style audio plug. These plugs connect to the microphone connectors on the front of the Larson-Davis microphone power supply, connecting the outdoor microphone units. For future designs, it is recommended that a BNC connector be used for the audio plug in to cut down on electrical noise in the connection. Both audio boards also have an external LED connected across the relay power. This LED indicates when the calibrator for that microphone is turned on.

The audio input board also connects to the field unit's CPU using a 14-pin ribbon cable. This ribbon cable connects the wind meter signal lines and the calibrator control lines to the card cage, as well as the +24-V power supply to drive the

calibrator relay. Because the two audio input PCBs are exactly identical (both accept the calibration signal on pin 12 of the 14-pin header), a twist in the cable is used to connect the calibration signal for the second microphone to the correct pin on the header. The audio input board before the twist should be attached to the first (top) microphone; the audio input board after the twist should be attached to the second (bottom) microphone. A schematic for the audio input board is shown as Figure E3.

Modified 24-V board

The monitor uses a 24-V board and power supply almost identical to the 24-V board used by the old monitors. The only significant difference between the 24-V board for the existing field units and the new field units is the addition of an LED after the audio relay driver transistor. Because this line is not used for any other purpose, this transistor is used to drive a status LED. The LED attached to the 24-V board indicates the current activity of the unit. If the unit is operating properly, the LED will blink at a rate of once per second. When the unit is in data collection mode, the LED will be on for half a second and off for a half a second. When the unit is in standby mode (with data collection disabled), the LED blinks on briefly once a second; when the unit's modem is connected, the LED blinks off briefly once a second. If the unit is in program upload mode, the LED blinks at a rate of once every 2 seconds. Figure E4 is a schematic of the 24-V board indicating the correct location for the extra LED.

Control wiring harness

The two audio input PCBs, the debouncer circuit, the 24-V board, and the card cage are all connected by a custom wiring harness. This wiring harness is a 14-wire ribbon cable attached to several different connectors: (1) one male D-shell connector for the parallel port on the CPU board, (2) another male D-shell connector that was originally attached directly to the watchdog board (the wind meter input) but is now connected to a debouncer circuit instead, (3) two 14-pin headers for the audio boards, and (4) a 14-pin Berg connector for the 24-V board. The debouncer D-shell connects lines 10 and 7 from the ribbon cable to pins 20 and 19 respectively. A 470-ohm resistor is placed between line 10 and pin 20 to prevent excessive current flow from the wind meter into the isolators on the debouncer board. Several other lines that control devices attached to the 24-V board (the heater, fan, and status LED) as well as the microphone calibrators are attached to a second D-shell and connected to the parallel port. This ribbon cable then attaches to the 24-V board and to the two audio input boards.

Since the two audio input boards are identical, lines 8 and 9 are reversed between the connections for the two boards. This allows the calibrator for the two microphones to be controlled individually. In the prototype unit, lines 7 and 10 are reversed as well. Since lines 7 and 10 are the wind meter input lines, their reversal means that the wind meter can only be attached to the first (top) microphone. However, since the wind meter is attached only to the top microphone, this is not a problem.

Appendix E includes a wiring diagram showing the exact connections made to each connector on the ribbon cable. Figure E5 shows the connections between the wiring harness and the CPU board.

Debouncer circuit

The opto-mechanical wind meters use a photodiode, which is an intrinsically analog device, to detect the motion of the cups pushed by the wind. However, this photodiode does not make a sharp transition between on and off (it makes a smooth and somewhat elongated transition). Therefore, there is a transition period when the wind meter will not consistently read as either a 1 or a 0, but instead is seen to switch rapidly between them. This results in interrupts being triggered more quickly than they can be handled. The end result in testing was that the MS-DOS running on the field unit's control processor would become confused and become unable to write to the internal SRAM storage. Not only did this crash the monitor, but it occasionally resulted in corrupted data queues. These problems made it necessary to implement a circuit to turn the wind meter's gradual transitions into sharp digital transitions.

The debounce circuit was designed as a retrofit to solve this problem. The circuit consists of a Motorola MC14490 hex contact debounce IC, an input optical isolator, and two D-shells (one male, one female). It attaches between the D-shell for the wind meter output and the watchdog board that contains the wind meter inputs. Because the debounce circuit only passes a transition when it has been stable for approximately 10 μs, it will prevent the transitions from resulting in interrupt overloads that may crash the controller CPU. This also means that a transient being misread as a transition will not result in a spurious count, improving the accuracy of the wind-speed detection. Figure E6 is a wiring diagram for the debounce circuit.

Microphone subsystem

Externally, the field unit attaches to two microphones and a wind meter. Since the Larson-Davis outdoor microphones used for the prototype field unit are

designed to screw into a pipe or nipple, the microphones were assembled into a unit that includes both the microphone and a small, sealed NEMA* enclosure that contains a wiring bay. The two microphone units attach to the 12-pin Amphenol connectors at the bottom of the field unit. The wind meter's signal connection to the field unit is also made via the 12-pin connector for Microphone 1.

The wiring bay inside the microphone enclosures accept the cables connected to the field unit. In the prototype, two cables were used for Microphone 1: a three-wire connection (two signal wires and a shield) is used for the wind meter signals, and a second cable carries the audio signal and the power and bias supply lines for the microphone preamps. For Microphone 2, only the signal and power supply wire is connected. The connections from the field unit terminate at a patch panel inside the enclosure. From this patch panel, they attach to the two 5-pin connectors at the bottom of the microphone. Figure E7 is a wiring diagram of the required connections between the connector from the microphone assembly to the field unit.

Field Unit Assembly

Figure 2 is an overall picture of the installed field unit in Spragueville. The total height of the pole is roughly 12 m. The lower crossarm is roughly 6 m off the ground and the two crossarms are about 3-m long. The two microphones are aligned vertically, which is necessary for proper operation of the blast-detection algorithm. The main enclosure is mounted approximately 3-m off the ground, to deter vandalism. Power and the telephone line are brought in from a box mounted lower on the pole (not visible in this picture). Two cables — a 7-conductor shielded cable for the microphone and a 3-conductor shielded cable for the wind meter — go from the main unit to the upper microphone. The junction box under the microphone connects the wind meter data lines to the main unit. An additional AC cable pair runs from the wind meter into the main unit to provide power for the heating element in the wind meter (to prevent icing). A second 7-conductor cable runs into the second microphone. Both the first pair of cables (the upper microphone and the wind meter data cables) and the second microphone cable terminate in 12-pin AMP connectors that mate with the modified audio PCBs described above.

* NEMA = National Electrical Manufacturer's Association

Figure 2. Installed noise monitoring field unit at Fort Drum, NY.

The Figure 3 view of the upper microphone and wind meter shows the cabling running between the main box, the junction box under the microphone, and the wind meter.

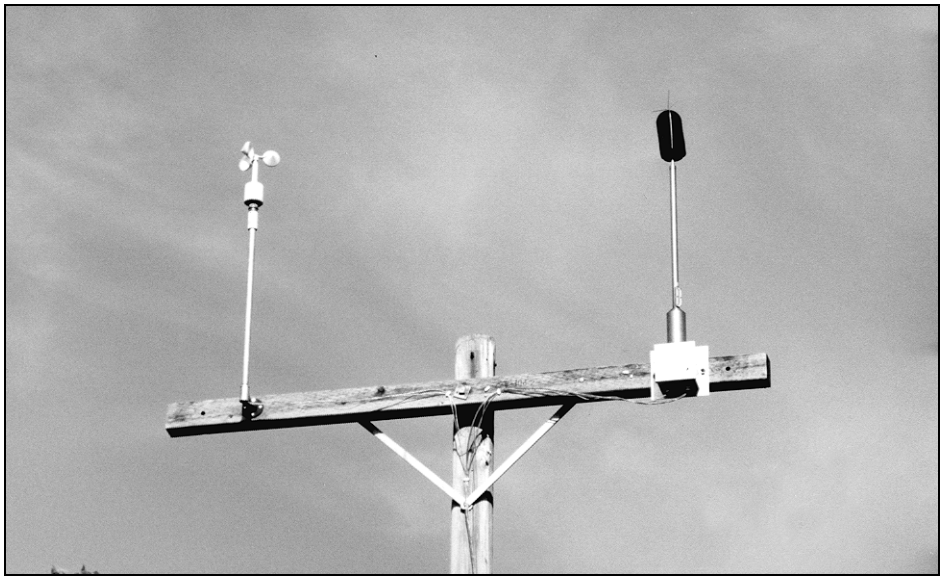


Figure 3. Closeup view of microphone and wind meter setup at Fort Drum, NY.

Figure 4 shows the inside of the main part of the field unit. This is a modified version of the original CERL field unit case. A full list of parts for the CERL Noise Monitoring and Warning System 98 is shown in Table 5 at the end of this chapter. The enclosure shown in Figure 4 has an air intake on the bottom left side, which opens into the duct on the left. Mounted in front of this duct is the power junction box (including the master power switch mounted on the top of the junction box) and the 120 VAC switched electrical outlet for the heater (the white box on the left). The heater is a standard, 1000-W, 120 VAC space heater. (Future designs may use alternate heaters because this heater is overly powerful and would be dangerous if it somehow became "stuck on.") On the top of the case, the mounting for the Hayes 300 bps modem has been removed and replaced with the Larsen-Davis microphone controller and preamp. This box provides a gain stage for the two microphones, along with the required bias voltages. (It also provides A- and C-weight filters, which should not be used.) The amplified and unfiltered output from the microphones is then sent to the DSP board via a BNC to 15-pin D-shell cable.

Visible by the end of the duct is a small board wrapped in black tape. This is the wind meter debounce circuit, which corrects the only real problem with the original design — that the wind meter's signal transitions were not sharp enough to ensure that only one edge was sensed when it changed state. The debouncer is based on a hex contact debounce IC and optoisolators and allows sharp transitions to be sent to the inputs. This prevents the crashes that occurred if a large burst of interrupts caused the stack to be overrun. This circuit connects to the 25-pin D shell that plugs into the watchdog board. It also connects to the watchdog board. The watchdog board conveniently provides a 56 kHz clock that was ideal for the debouncer IC.

Mounted to the right of the microphone preamplifier are a 120 VAC to 28 VAC transformer and a CERL 24-V board. The 24-V board is nearly unmodified (in fact, a completely unmodified 24-V board may be used in a pinch). The only change is that an LED, used to indicate that the field unit is working, is mounted on the board. (It is attached to the drive transistor for the modem-power relay, which is unused in this design.) This LED blinks at 1 Hz when the unit is operating. A shortened duty cycle indicates that data collection is disabled, and a lengthened duty cycle indicates that the modem is connected.

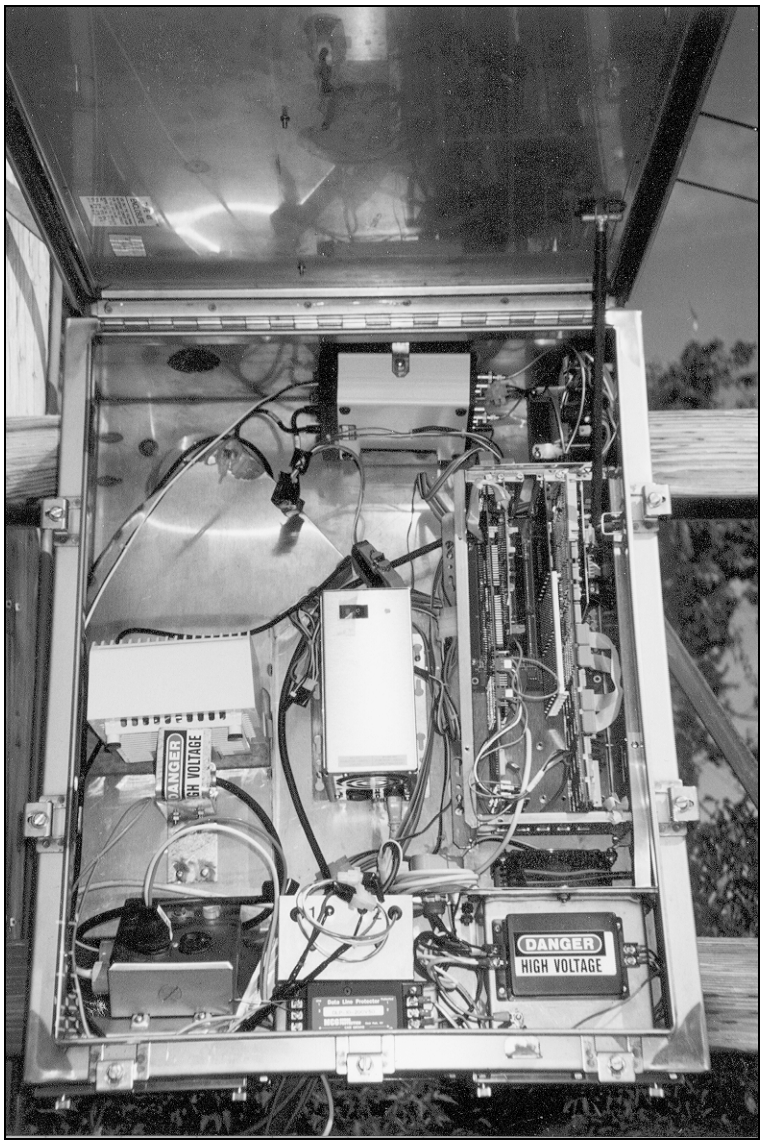


Figure 4. Internal view of the CERL Noise Monitoring and Warning System field unit case.

At the center of the case is the power supply, which is an ordinary computer power supply that provides +12 V, +5 V, -5 V, and –12 V DC to the rest of the system. The power switch should always be left ON (power can be cut off with the switch on the junction box, which also turns off the 24-V power supply and the rest of the live AC in the system). The 24-V board is connected into the control signal harness that terminates with the 25-pin D shell visible above the power supply and another shell attached to the leftmost board in the card cage.

Below the power supply is the box containing both audio interface boards, as well as the telephone line surge suppresser. The connections from the audio box are brought-out Molex connectors (used for the bias voltage and power lines) and eighth-inch signal connectors. (It probably would have been better to bring the

audio signal out on coaxial connectors because the noise floor of this field unit is higher than it should be.) The audio interface board is also connected to the control wiring harness.

On the right side of the box is the main card cage. Under the cage is the main exhaust fan and two AC line filters. The first AC line filter is the series filter, which is the box labeled "HIGH VOLTAGE." To the left and above this box is the second (parallel) filter, which is the surge suppresser in the gray plastic case with the cord wrapped around it. The cord is for the keyboard and is used for maintenance — a conventional AT-style, 5-pin keyboard can be plugged into it.

From left to right, the cards in the card cage are:

- CPU board
- Watchdog timer (WDT-501P) board
- Empty slot for a VGA board (leave empty when not in use to improve airflow)
- PCF-1 virtual disk SRAM/FLASH ROM board
- Dual DSP board
- 33.6 kbps USR modem.

These boards are all mounted in a passive ISA backplane, and connect to the rest of the field unit with the following connections:

- CPU board:
  - parallel port connects to the control wiring harness ("port 1")
  - one serial port connects to the console serial plug on the left side of the cage
- Watchdog: DB25 connects to the debouncer circuit board
- DSP: The front D shell (which connects to the A/D and D/A board mounted on the DSP) connects to the two microphone outputs on the Larson-Davis preamp.
- Modem: The telephone line connects through the phone surge suppressor mounted on the bottom of the field unit's case.

Table 6. Parts manifest and sources.

Part Number	Description	Source
SB4862PVN/66	i486/66 Single Board Computer	Industrial Computer Source
OEMC-06	6-slot ISA card cage	Industrial Computer Source
OEMC-P15	150-W A/C computer power supply	Industrial Computer Source
FD-1.4M	3.5 inch 1.4mb floppy disk drive	Industrial Computer Source
WDT-501P	Watchdog Timer Board	Industrial Computer Source
PCF1-0MFS	PCF-1 SRAM/FLASH Disk Emulator Board	Industrial Computer Source
PCF1-FE1	1 MB FLASH EPROM SIMM	Industrial Computer Source
PCF1-SR1	1 MB SRAM SIMM	Industrial Computer Source
USR-000840-00	33.6 kbps USR Sportster Internal Modem	Computer Discount Warehouse
	DPC/C40B Dual DSP carrier board	Spectrum Signal Processing
	2 x TMS-32C040 40MHz/384K module	Spectrum Signal Processing
	AM/D16DS Crystal ADC/DAC Daughter Module	Spectrum Signal Processing
	DPC/C40B DSP port cables	Spectrum Signal Processing
2200C	Microphone Preamp/Power Supply	Larson-Davis
	Microphone Bases and Cabling	CERL
5830-00-R10-3677	2 x Outdoor Microphone	Larson-Davis
Cable A	Audio Cables (BNC-26 pin D shell)	CERL
19435F	Fan	EG&G Rotron
	Heater	Ace Hardware
84-01-602	Modified 24-V board (with added LED)	CERL
P-8664	120 VAC-28 VAC transformer	Stancor
DLP-10-200V50	Telephone Line Protector	MCG Electronics
J9200-10	Parallel Surge Protector	Square D Company
I-102	Parallel Power Isolator	Control Concepts
84-01-703	2 x Audio Input Boards	CERL
Cable B	Audio Input Harness	CERL
Cable C	24-V Cable Harness	CERL
	Wind meter Debounce Circuit	CERL
A-30H240855LP	Noise Monitor Enclosure (modified)	CERL (Hoffman modified)
276-452L	120-V electrical box	Bell Electrical
Maintenance Parts		
SVGA2	SVGA video adapter (generic)	Industrial Computer Source
FD-1.4M	1.44 mb floppy disk drive (generic)	Industrial Computer Source
KB2	101-key keyboard (generic)	Industrial Computer Source

6 Conclusions and Recommendation

The CERL Noise Monitoring and Warning System 98 appears to enhance the signal-to-noise ratio by about 10 dB. In effect, it can measure blast sounds reliably at a level 10 dB lower than could the original CERL noise monitor. Chapter 5 on implementation shows that the monitor is technically feasible. It is recommended that the new Noise Monitoring and Warning System 98 be transferred to the field via a suitable DEM-VAL program such as ESTCP.

References

Benson, Jonathan W., “A real-time blast noise detection and wind noise rejection system,” *Noise Control Engineering Journal*, 44 (6), Nov-Dec 1996.

Texas Instrument TMS320C4x User's Guide 2, SPRU063C (Dallas, Texas).

Appendix A: Console Commands

This list of commands can be issued to the CERL Noise Monitoring and Warning System 98, in console mode. All of this information is available online. Use "SHOW COMMANDS" to show the list of available commands, and "HELP <command>" to show the definition and usage of a particular command.

QUIT

Syntax: ?QUIT
QUIT shuts down the Noise Monitor control program. Under normal circumstances, it will restart automatically.

KERMIT

Syntax: ?KERMIT [MODEM | CONSOLE]
KERMIT shuts down the noise monitor and runs a KERMIT server on the underlying hardware. This allows maintenance and updates to the noise monitor's control software from over a serial port or through the modem. The noise monitor will normally restart automatically when the modem hung up or the serial cable is unplugged.

KERMIT MODEM sets up KERMIT to communicate to the system's modem.

KERMIT CONSOLE sets up KERMIT to communicate via the console serial port.

HELP

Syntax: ?HELP [COMMAND]
HELP, used alone, provides a general summary of the help options available for the Noise Monitoring and Warning System.
To find help on a particular command, use "HELP <command>".

START

Syntax: ?START [AT <time>] [FOR <length>] [CAL]
START is used to take a manually collected data block.
To start an indefinite-length data block immediately, simply type "START". STOP will end the data block and record it.
To take a data block of a fixed length, use "START FOR <length>."
The data block will be ended automatically when the time is up.

To take a future data block, specify AT <hh:mm:ss> [<mm/dd/yyyy>]; the day is assumed to be today if not specified. To cancel a future data block, use “START CANCEL”.
Add “CAL” to the end of any START command to request that the calibrators be enabled during the sample.

STOP

Syntax: ?STOP
STOP cancels any manual data blocks currently in progress. Any manual data block in progress will be immediately terminated, and the SEL and PEAK values will be recorded from the start up until that point.

STATUS

Syntax: STATUS [ON|OFF]
The STATUS command displays the information that would normally be displayed on the status line at the top of the screen in an expanded format. “STATUS ON” enables this status line on remote consoles, “STATUS OFF” disables it for slow links.

```
                                        Flags         Wind Speed      o
LCSEL   36.9 LFPK   51.0 | PK   50.9 |       |    1 |   0 MPH |  97.9  PS OK
CSEL of last block taken  Peak of last tenth-second sample      Internal
Status
        Flat peak of last block taken    Number of blocks in queue
```

Flags are: C - Calibration block in progress M - Manual block in progress

T - Threshold block in progress
B - Blast Detected W - Windy conditions
O - Online to a base station
S - Standby mode

A + after the number of blocks indicates that one or more blocks in the queue are “important,” and that the Base Station will be called when possible.

CAL

Syntax: ?CAL [CH1 | CH2 | BOTH | IMM | IMM1 | IMM2 | IMMB]
CAL starts a calibration.
CAL starts a calibration of whatever microphones are currently being used to collected data. CAL CH1, CAL CH2, and CAL BOTH start a normal calibration of the specified microphones.
CAL IMM immediately takes a calibration block from the active microphones, without turning on the calibrators or waiting for them to stabilize. Normally this should not be done. IMM1, IMM2, and IMMB calibrate the first, second, or both microphones.

SET

Syntax: ?SET <variable> [<parameters>]
SET <variable>, without further parameters, displays the value of a variable.
SET <variable> <parameters> sets the value of a given configuration variable.
SHOW OPTIONS lists the known variables for the SET command;
HELP SET <variable> gives the possible settings and more information about the variable.

SHOW

Syntax: ?SHOW [BLOCK
[n] | THRESH | FILTER | OPTIONS | COMMANDS | <option name>]
SHOW COMMANDS lists the available commands.
SHOW OPTIONS lists the available configuration variables.
SHOW <variable names> shows the current value of a variable.
SHOW BLOCK shows the last data block taken.
SHOW BLOCK <n> shows the nth-oldest data block in the queue.
SHOW THRESH shows settings that control when blocks are taken.
SHOW FILTER shows settings that control which blocks are rejected or considered to be important.

ENABLE

Syntax: ?ENABLE
ENABLE turns off the flag that disables threshold data collections during the microphone warm-up period and microphone calibration. Normally, this command need not be used except for testing. (Note that it does not enable or disable threshold mode; use SET THRESH_MODE to do that.)

DUMP

Syntax: ?DUMP [ALL]
DUMP discards the oldest data block in the queue.
DUMP ALL discards all of the data blocks in the queue.
Use these commands only if you are certain that no valid data is stored in the noise monitor's data queue.
DUMP CONFIG resets the configuration of the noise monitor to the defaults contained in the FLASH ROM in the field unit.

HANGUP

Syntax: ?HANGUP
HANGUP forces the noise monitor to immediately disconnect any incoming call on the modem. It can be used to hang up an incoming console session.

CALL

Syntax: ?CALL
CALL forces the noise monitor to call its programmed base station number at the next available opportunity.

REDIRECT

Syntax: ?REDIRECT CONFIG | DATA <path>

?REDIRECT LOCK
REDIRECT CONFIG <path> tells the noise monitor to read in its configuration files from the specified path instead of the current directory. (The redirect is stored in a configuration file in the current directory.) Normally, the configuration will be redirected from the ROM disk into the RAM disk.
REDIRECT DATA <path> tells the monitor to read its data files from the specified path instead of the current directory.
REDIRECT LOCK tells the monitor to not allow further redirection.
Neither of these redirections takes place until the system is restarted.
If no configuration files are found in the specified path, they will be created based on the defaults stored in the current configuration files.

DIE!!

Syntax: ?DIE!!
DIE!! tests the watchdog timer by doing a 3-second delay in the program code. The monitor should restart automatically when this command is issued; if it does not, the failure of the watchdog timer will be reported and the monitor will be restarted.

VERSION

Syntax: ?VERSION
Displays noise monitor version information and the time since the last reboot.

SEND

Syntax: ?SEND <string>
Sends a literal string to the modem.
Note: Due to the internal architecture of the code, the string will be converted to upper case first. This will not normally be a problem.

Appendix B: Monitor Options

This is a list of user-settable options that control the noise monitor operation. All of these parameters can be set online; use “SET <option> <value>.” “SHOW <option>” shows the current value of an option. Like the information for commands, all of this information can be viewed online. Use “SHOW OPTIONS” for a list of available options, and “HELP SET <option>” for more information on a given option.

THRESH_MODE

Syntax: ?SET THRESH_MODE ON|OFF
THRESH_MODE enables or disables threshold data collection.

FPK_THRESH

Syntax: ?SET FPK_THRESH <dB>|OFF
FPK_THRESH sets the threshold on the flat-weighted peak over which a threshold data block will be started. The value is given in calibrated decibels. Normally, this value will not be set. (Use “OFF” to disable the threshold.)

CPK_THRESH

Syntax: ?SET CPK_THRESH <dB>|OFF
CPK_THRESH sets the threshold on the C-weighted peak over which a threshold data block will be started. The value is given in calibrated decibels. Normally, this value will not be set. (Use “OFF” to disable the threshold.)

FSEL_THRESH

Syntax: ?SET CPK_THRESH <dB>|OFF
FSEL_THRESH sets the threshold on the flat-weighted SEL for a tenth-second block over which a threshold data block will be started. The value is given in calibrated decibels. Normally, this value will not be set. (Use “OFF” to disable the threshold.)

CSEL_THRESH

Syntax: ?SET CSEL_THRESH <dB>|OFF
CSEL_THRESH sets the threshold on the C-weighted SEL for a tenth-second block over which a threshold data block will be started. The value is given in calibrated decibels. Normally, this value will not be set. (Use “OFF” to disable the threshold.)

BLAST_THRESH

Syntax: ?SET BLAST_THRESH <number>|OFF
BLAST_THRESH sets the minimum number of blast-detection blips that must be detected in a tenth-second block to trigger a threshold block. The higher the number, the more likely it is to ignore a windy block, but it also increases likelihood of missing a quiet blast.
SET BLAST_THRESH OFF to disable.

ABS_THRESH

Syntax: ?SET ABS_THRESH <number>|OFF [dB|dB ABS]
ABS_THRESH is the threshold used by the base stations. It is normally set in terms of a number from 33000-65535 that represents a flat peak level coming in from the microphone sampler; however, it can also be set in terms of calibrated or absolute decibels. (Use “SET ABS_THRESH <number>” to set it normally; use “SET ABS_THRESH <number> dB” to set it in decibels, or “SET ABS_THRESH OFF” to disable the absolute threshold. “SET ABS_THRESH <number> dB ABS” sets the absolute threshold in terms of uncalibrated peaks.)

FPK_FILTER

Syntax: ?SET FPK_FILTER <dB>|OFF
If set, the FPK_FILTER is the minimum peak level for a block that will not be subjected to the KEEP_FILTER flag. Normally, this will be set above the threshold and used to prevent calls from being made to the base station for minor noises.

CPK_FILTER

Syntax: ?SET CPK_FILTER <dB>|OFF
If set, the CPK_FILTER is the minimum C-weighted peak level for a block that will not be subjected to the KEEP_FILTER flag. Normally, this will be set above the threshold and used to prevent calls from being made to the base station for minor noises.

FSEL_FILTER

Syntax: ?SET FSEL_FILTER <dB>|OFF
If set, the FSEL_FILTER is the minimum SEL for a block that will not be subjected to the KEEP_FILTER flag. Normally, this will be set above the threshold and used to prevent calls from being made to the base station for minor noises.

CSEL_FILTER

Syntax: ?SET CSEL_FILTER <dB>|OFF
If set, the CSEL_FILTER is the minimum C-weight SEL level for a block that will not be subjected to the KEEP_FILTER flag. Normally, this will be set above the threshold and used to prevent calls from being made to the base station for minor noises.

LENGTH_FILTER

Syntax: ?SET LENGTH_FILTER OFF|<maximum length> [minimum length]
LENGTH_FILTER allows the KEEP_FILTER flag to be applied to samples that exceed a given maximum length or are shorter than a given minimum length. (Lengths are given in seconds.)

KEEP_FILTER

Syntax: ?SET KEEP_FILTER [ON|OFF|CALL]
This flag controls the disposition of blocks that do not fall within the bounds of the Noise Monitor filters. (Use SHOW FILTER for a list of these filters.)
If KEEP_FILTER is ON, then these blocks will be kept.
If KEEP_FILTER is OFF, then they will be discarded.
If KEEP_FILTER is CALL, then data blocks that do not pass the filter will still be considered important and will trigger calls to the base station.
If KEEP_FILTER is BLAST, then data blocks that do not pass the filter will be kept if they are identified as blasts by the blast detection algorithm. They will not be considered important.
If KEEP_FILTER is CALL-BLAST, then data blocks that are identified as blasts will be kept and considered important regardless of their having passed the filter. Otherwise, they will be discarded.

CAL_CONST

Syntax: ?SET CAL_CONST CH1|CH2 <value>
CAL_CONST is the adjustment from decibel levels reported by the DSP board to actual acoustic decibels. Normally, it is adjusted automatically as part of the calibration process. To set it explictly, use “SET CAL_CONST CH1 -56” to set the Channel 1 calibration constant to -56 dB.

CAL_LVL

Syntax: ?SET CAL_LVL CH1|CH2 <value>
CAL_LVL sets the calibrator level, which is the acoustic sound level (in decibels) that an active calibrator corresponds to. Piston phone calibrators are usually 124 dB; the internal calibrators on microphones are usually around 90 dB. To set CH1 to assume a piston phone calibrator, use "SET CAL_LVL CH1 124."

CALIBRATOR

Syntax: ?SET CALIBRATOR [CH1|CH2] ON|OFF
SET CALIBRATOR turns the microphone calibrators on and off, and enables or disables threshold sampling appropriately. To turn the CH1 calibrator on, type "SET CALIBRATOR CH1 ON"; to turn both off, type "SET CALIBRATOR OFF."

4184

Syntax: ?SET 4184 ON|CAL|OFF
4184 controls a special mode for the B&K 4184 microphone, which uses acoustic calibrators instead of electrostatic ones and needs special treatment. Normally, these calibrators are not used for calibration, but instead are used as a self-check to make sure that the microphones are functioning properly. In addition, the calibrations need to be staggered, because they can interfere with each other.
"SET 4184 ON" enables the staggered calibration and disables setting calibration constants from the calibration blocks.
"SET 4184 CAL" enables staggered calibration, but sets calibration constants.
"SET 4184 OFF" disables the 4184-specific modes, for electrostatic calibrators.

USE_CAL

Syntax: ?SET USE_CAL ON|OFF
Disabling USE_CAL displays uncalibrated values for the peak and SEL values on the status line. Enabling it shows calibrated values for peak and SEL.

PRETRIG

Syntax: ?SET PRETRIG <seconds>
PRETRIG is the number of seconds that are read into the threshold block when the threshold is exceeded, in tenth-second blocks. Up to 4 seconds can be included before the actual block that exceeded the threshold.
"SET PRETRIG 0.5" sets the pretrigger time to 0.5 sec.

POSTTRIG

Syntax: ?SET POSTTRIG <seconds>
POSTRIG is the number of seconds that are read into the threshold block when the threshold is exceeded, in tenth-second blocks.

VERBOSE

Syntax: ?SET VERBOSE [ON|OFF]
VERBOSE controls how much information is displayed when a block is taken. "SET VERBOSE OFF" displays a single line with the length of each block as it is taken; "SET VERBOSE ON" displays the peak and SEL data for each block as well.

MODEM_PORT

Syntax: ?SET MODEM_PORT 1|2
MODEM_PORT controls which COM port (1 or 2) the noise monitor expects to find the modem on. This will not normally need to be changed.

MODEM_SPEED

Syntax: ?SET MODEM_SPEED [VAR|PAC] <speed>
This sets the speed at which the noise monitor talks to the modem. If "VAR" is specified, the rate given is the maximum rate that will be used; the port speed will be reduced based on the CONNECT message. (This is needed for compatibility with the base station.) If "PAC" is specified, pacing will be enabled, and data will be sent to the modem at a rate no greater than the actual connect speed.
Valid modem speeds are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600

CONSOLE_PORT

Syntax: ?SET CONSOLE_PORT <port>
Sets the port number for the serial console (1 or 2).
This will not normally need to be changed.

CONSOLE_SPEED

Syntax: ?SET CONSOLE_SPEED <speed>
Sets the port speed for the serial console.
This will not normally need to be changed.
Valid speeds are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600

PORT_LOCK

Syntax: ?SET PORT_LOCK ON
This locks all of the port settings, including MODEM_PORT, MODEM_SPEED, CONSOLE_PORT, CONSOLE_SPEED, and INIT. Once they have been locked, they cannot be unlocked and cannot be changed.

PHONE

Syntax: ?SET PHONE [ON <phone number>|OFF]
SET PHONE ON <phone number> sets the telephone number to be dialed when the noise monitor collects a data block that is considered "important."
(Use 'T' in the telephone number to switch to tone dialing, and 'P' to switch to pulse. Tone dialing is the default.)
SET PHONE OFF tells the monitor NOT to call into a base station when a data block is collected.

CALL_THRESH

Syntax: ?SET CALL_THRESH <number>
CALL_THRESH sets the number of items that can be stored in the data queue before the base station places a call. As long as this number is not exceeded, no call will be placed until an "important" data block istaken.

MODEM_INIT

Syntax: ?SET MODEM_INIT [ON <init string>|OFF]
Normally, the modem init string would never be disabled.
"SET MODEM_INIT ON ATZ" tells the modem to reset to its default configuration on startup.
"SET MODEM_INIT OFF" disables modem initialization, which is a bad idea because the modem cannot adjust its baud rate to match that of the computer.

UNIT

Syntax: ?SET UNIT <number>
SET UNIT <number> sets the noise monitor's serial number as seen by the base station. Valid range is 0-63. "SET UNIT LOCK" locks the unit number against future changes.

CHANNEL_MODE

Syntax: ?SET CHANNEL_MODE CH1|CH2|MEAN
CHANNEL_MODE controls which microphones' data are used in the SEL and PEAK data sent to the noise monitor. Normally, a geometric mean of the two channels is sent; however, if one microphone is less reliable than the other, the data sent to the monitor can be set to come from either one.

CAL_INTERVAL

Syntax: ?SET CAL_INTERVAL <hours>
This option sets the number of hours between automatic calibrations.
An autocal interval of 0 disables automatic calibrations.
Note that automatic calibrations will always occur on startup.

SAMPLE_TIMEOUT

Syntax: ?SET SAMPLE_TIMEOUT [OFF|<secs>]
This option sets the maximum length of a threshold-mode data sample.
The sample will be cut off at the timeout; if the threshold is exceeded
again, a new sample will be started.

KEEP_WINDY

Syntax: ?SET KEEP_WINDY CALL|ON|OFF
If KEEP_WINDY is ON, then windy blocks will be kept.
If KEEP_WINDY is OFF, then windy data blocks will be discarded.
If KEEP_WINDY is CALL, then windy data blocks will still be considered
important, and will trigger calls to the base station.
If KEEP_WINDY is BLAST, then data blocks that are marked as windy will
be kept only if they are identified as blasts by the blast detection
algorithm. They will not be considered important.
If KEEP_WINDY is CALL-BLAST, then data blocks that are identified as
blasts will be kept and considered important regardless of the windy flag.
Otherwise, they will be discarded.

KEEP_BAD

Syntax: ?SET KEEP_BAD CALL|ON|OFF
If KEEP_BAD is ON, then data blocks not identified as having blasts
will be considered unimportant (a call will not be made when one is
collected), but it will be stored in the queue and sent when the
unit connects to a base station.
If KEEP_BAD is OFF, then data blocks not identified as having blasts
will be discarded.
If KEEP_BAD is CALL, then data blocks not identified as having blasts
will still be considered to be important, and a call will be placed to warn
the base station of them.

WIND_SAMPLE

Syntax: ?SET WIND_SAMPLE <seconds>
WIND_SAMPLE controls how long the wind meter accumulates ticks before
it reports the speed back to the monitor. It must be less than the
WIND_TIMEOUT for the WINDY flag to be meaningful.

WIND_THRESHOLD

Syntax: ?SET WIND_THRESHOLD <ticks>
WIND_THRESHOLD sets the number of pulses that must be received from the wind meter in the sample time in order to set the WINDY flag. Normally, you would set this in terms of miles per hour; use WIND_MPH instead.

WIND_TIMEOUT

Syntax: ?SET WIND_TIMEOUT <seconds>
WIND_TIMEOUT controls how long the data collected is considered WINDY after a block in which the number of ticks read exceeds the threshold. Normally, this timeout period will be longer than the WIND_SAMPLE period.

WIND_MPH

Syntax: ?SET WIND_MPH <MPH>
WIND_MPH sets the WIND_THRESHOLD to the correct number of pulses to match up with the miles per hour specified, based on the current wind meter type and the wind sample interval.

WINDMETER

Syntax: ?SET WINDMETER <windmeter type>|<mult add>
SET WINDMETER <windmeter type> reads the WIND.TXT file for the given Wind meter type, and sets the Multiplier and Add parameters appropriately.
SET WINDMETER <multiplier> <add> sets a custom wind meter type, with the given Add and Multiply parameters.

TIME

Syntax: ?SET TIME <hh:mm:ss>
SET TIME <hh:mm:ss> sets the noise monitor time to the given time.

DATE

Syntax: ?SET DATE <dd/mm/yyyy>
SET DATE <dd/mm/yyyy> sets the noise monitor date to the given date.

CALL_THRESH:

Syntax: ?SET CALL_THRESH <number>
This sets the unit to call out to the base station to empty its queue when at least <number> events are in the queue. (This ensures that the queue does not overfill and lose events or tie up the phone for hours when an important block finally does arrive. At 300 bps, emptying the 2047-element data queue can take a while.)

REBOOTS

Syntax: ?SET REBOOTS [RESET]
SET REBOOTS or SHOW REBOOTS shows the number of times the system has been booted since the counter was last cleared.
SET REBOOTS RESET clears the boot counter.

HEATER

Syntax: ?SET HEATER [ON|OFF]
SET HEATER ON enables the heater inside the case of the noise monitor, and SET HEATER OFF turns it off.
(Note that, if the internal temperature is above 20 °C, it will be immediately turned off again by the internal control program. Likewise, if the internal temperature is below 12 °C, the heater will immediately be turned on.)

FAN

Syntax: ?SET FAN [ON|OFF]
SET FAN ON enables the fan to bring air from the outside into the case of the noise monitor, and SET FAN OFF turns it off.
(Note that, if the internal temperature is below 20 °C, it will be immediately turned off again by the internal control program. Likewise, if the internal temperature is above 30 °C, the fan will immediately be turned on.)

Appendix C: Field Unit Software Source Code

Controller Source Code

The controller source code is written for MS-DOS, using the Borland C++ 3.1 compiler. Although it is written using C++ extensions, the vast majority of the code is procedural C code. The only C++ extension used extensively is the "//" style comments. However, the DSP code (DSP.CPP) is written using classes because of the code inherited from the previous developer. Although there are a few classes, they do not really represent objects. It was simply most convenient for the developer to leave the existing structure of this code unchanged when work began on it. Unfortunately, this resulted in the DSP.CPP module being by far the most cryptic part of the control code.

This appendix contains the source code to the field unit's control program. Each core file is preceded by annotations listing the functions contained in that source file, and other information useful in understanding what that module does and how it relates to the other modules and the program as a whole.

CMD.CPP and CMD.H

CMD.CPP and CMD.H make up the field unit's command handler. In the header file is a list of text and symbolic names for every command and option ("SET") known to the field unit. The symbolic command names are declared in the COMMANDS enumerated type ("enum COMMANDS"); the text names are stored in the command_strings array (char *command_strings[]). Likewise, the symbolic option names are declared in the OPTIONS enumerated type (enum OPTIONS), and the text names are stored in the option_strings array (char *option_strings). It is important that the corresponding lists have the same number of elements, and that each element in the two lists match up. If the symbolic names and text names do not match up, the commands will be parsed incorrectly. The string arrays are terminated with a null pointer. If new commands are to be added, add them to the ends of the list before the "END" tag and the null pointer.

CMD.CPP contains the following functions:

char *parse_cmd(char *cmd, enum COMMANDS *command)

parse_cmd identifies the command input as the first word pointed to by cmd, and returns the command index in *command. It also returns the parameters (second and later words) as the return value. If the command is invalid, it returns CMD_INVALID (-1) in *command.

char *parse_set(char *cmd, enum OPTIONS *option)

parse_set is similar to parse_cmd, but works for options. The option is presumed to be the first word pointed to by cmd, and the returned parameters are the second and later words. If the option is invalid, it returns OPT_INVALID (-1) in *command; otherwise, it returns the option index in *command.

void do_command(char *cmd)

do_command accepts a command from the console in *cmd, and executes the command. This function contains all of the code necessary to act on the commands. It uses parse_cmd to identify the command being issued and then uses a switch statement to execute the correct instruction code.

void do_set(char *cmd)

do_set is called by do_command to act on a “SET” command. It is passed in the parameters to the SET command, then uses parse_set to identify the option, and then acts on it using a large switch statement.

int init_help(char *path)

init_help initializes the help system by scanning the help files and finding the locations of the help for all of the commands and options in that file.

void printHelp(enum COMMANDS command, char *params)

printHelp displays the help for the command identified by “command.” If command is CMD_SET, it uses the “params” string to identify which option value to print the help for. (If CMD_INVALID is passed in, the generic help will be provided. Also, if command is CMD_SET and params is non-empty but does not contain a valid option name, the generic option help will be provided.)

The help system reads two files, CMD_HELP.HLP and OPT_HELP.HLP, stored with the program code on the FLASH ROM. (The init_help command is called with an empty path, which will simply access the current directory. This is "A:\" on the monitor.) CMD_HELP.HLP stores the help information for commands. Information from it is displayed when the user types "HELP <command>" OPT_HELP.HLP stores the help information for settings. Information from it is displayed when the user types "HELP SET <option>".

These files start with generic help, which is displayed when no specific help is available or the input is not a valid command or setting, and are followed by specific help on various settings. These specific help sections are preceded by lines of the form:

>*setting or option name*

where "setting or option name" is the name given in the command_strings or option_strings array. A line of the form:

>*END

must be the last line of the file.

DATACONV.CPP and DATACONV.H

This module provides conversions to and from the ASCII data formats used by the existing field units, and is used for compatibility with the existing base stations. The formats that this converts to and from are defined by the field unit protocol specification. This means writing each four bits as an ASCII value between 0x20 and 0x2F, least significant bit (LSB) first (little endian), except where noted. It defines the following functions:

Note that all of the conversions to ASCII format return 1 for success and 0 for failure. (Many of them always return 1 and cannot fail.)

int conv_byte2ascii (unsigned char byte, char *string)

Converts 1 byte (an 8-bit unsigned number) into the 2-character string used by the original CERL field units.

int conv_int2ascii(unsigned int word, char *string)

Converts one word (a 16-bit unsigned number) into the 4-character string used by the original CERL field units.

int conv_date2ascii(struct date *datep, char *string)

Converts a date (stored in the datep structure) into the 4-byte ASCII date form used by the original CERL field unit. This is done by first converting the date to an integer using conv_date2int() then converting the integer to a string using conv_int2ascii. This function is Y2K compliant.

int conv_time2ascii(struct time *timep, char *string)

Converts a time (stored in the timep structure) into the 6-byte ASCII form used by the original CERL field unit.

int conv_peak2ascii(float peak, char *string)

Converts a peak level, as returned by the DSP code (which is really 65536 times the peak level returned by the DACs), into the ASCII format used by the original monitor. The peak is converted to an unsigned integer by dividing by 65536 and finding the absolute value, then converting that integer to a 2-character ASCII string.

int conv_sel2ascii(float sel, char *string)

This converts the Sound Exposure Level stored in SEL to the 16-byte character string (storing an 8-byte integer) used by the original monitors. This is done by first dividing out 2^32 (65536^2), then converting the floating point number into an integer by repeatedly taking out the first 4 bits.

int conv_len2ascii(unsigned long len, char *string)

This converts from the number of tenth-second blocks taken into the number of 50-ms blocks taken, then sends the resulting number as a 12-byte string.

int conv_pp2ascii(unsigned long pp, char *string)

This converts from the peak position stored by the monitor (which is in terms of 0.5 ms “microblocks”) into the peak position used by the old monitor (which is in terms of its sample rate of 20 kHz), then returns the resulting number as an 8-byte ASCII string.

int conv_byte2bcd(unsigned char byte, char *string)

Converts a number between zero and 99 into the binary-coded decimal format (2 bytes) used by the original field units.

Both digits are recorded (least significant digit first, then most significant digit) as ASCII values between 0x20 and 0x29.

The following functions convert from ASCII forms to the forms used by the field unit internally:

int conv_ascii2byte(char *string)

Returns the number (0-255) represented by the first two bytes of "string".

unsigned int conv_ascii2int(char *string)

Returns the unsigned integer (0-65536) represented by the first four bytes of "string".

int conv_ascii2date(char *string, struct date *datep)

Returns the date represented by "string" in the "datep" structure.

int conv_ascii2time(char *string, struct time *timep)

Returns the time represented by "string" in the "timep" structure.

float conv_ascii2peak(char *string)

Converts the ASCII peak value into a floating point peak value that can be used by the monitor. This is done by converting the input string to a 16-bit unsigned integer, then returning that number, times 65536, as a floating point number.

float conv_ascii2sel(char *string)

Converts a 12-byte ASCII string representing a SEL back into a floating point number. This is done by first converting the string into a floating point number (starting with the LSB and accumulating upward to the end of the number), then mulitplying the result by 2^32. Note that the 32-bit floating point number format cannot exactly represent any arbitrary number written in this form.

int conv_ascii2int3(char *string)

Converts a 3-character (12-bit) number stored in ASCII form to an unsigned integer.

int conv_bcd2byte

Converts a Binary Coded Decimal (BCD) ASCII string into a number between 0 and 99. This is done by adding the first digit (minus 0x20) to 10 times the second digit (minus 0x20).

The following two functions can be used to convert the encoded date format (recorded as days after January 1, 1978) to and from the DOS "date" structure. Note that this form is good until about 2067, so there are no Y2K issues with this code.

unsigned int conv_date2int(struct date *datep)

Converts the DOS date structure "datep" into an integer day (in days since January 1, 1978).

void conv_int2date(unsigned int datenum, struct date *datep)

Converts the integer number of days since January 1, 1978 into the DOS date structure "datep."

DSP.CPP and DSP.H

DSP.CPP is probably the most complex part of the field unit's control software. DSP.CPP contains the code to tenth-second blocks from the DSPs, detect threshold conditions, and accumulate tenth-second blocks into manual and threshold data blocks.

DSP.CPP is largely based on code inherited from a previous developer. Therefore, it has a somewhat different structure than the other modules. It consists of several classes; however, these classes should really be thought of as structures. The structures are all "friends" of each other (so they have access to each other's private data) and very little data abstraction is actually done by the classes.

1? a degenerate class "Sample," which matches the in-memory structure of the tenth-second blocks collected by the DSPs and stored in the dual-port RAM.

2? "Block," which accumulates the "Sample" blocks into larger blocks to create manual input data blocks or threshold blocks.

3? "tDSP," which defines the interface between the DSP board and controls sampling and block formation

4? "tCommand," a command priority queue to sequence operations.

DSP.CPP has a local function used for timing:

time_t dtime(time_t *timer)

dtime returns the unix-format time (seconds since January 1, 1970) and puts it into *timer (if *timer is not NULL.)

The Block class uses the following functions:

Block::Block(void) (the constructor)

This initializes a few of the variables of the block structure. However, it doesn't clear out the actual data accumulation variables — Block::clear(void) does that.

void Block::clear(void)

This finishes the initialization of the Block by resetting all of the block parameters: the block length, the sampling flag, the windy flag, and all of the variables that accumulate the tenth-second block data sent by the DSP.

void Block::start(void)

This initializes a manual, untimed (STOP/START) sample in the Block by recording the start time and date, the type, and then marking that sampling is occurring.

void Block::cstart(int ch)

This initializes a calibration in the Block for the channel specified by ch (either 1 or 2, for the top or bottom microphone respectively) by setting the sample time (100 block or 10 seconds), the current time, the type, and the sampling flag to correspond to the calibration of either or both microphones. If ch is negative (-1, -2, or -3) cstart begins a background check block. The background check is used to verify that ambient noise is not too much for an accurate calibration.

void Block::istart(int len)

This starts a manual, timed sample in the Block by recording the length, start time, date, and type, and by marking the sample as occurring.

void Block::record(tDSP *DSP)

This function is primarily responsible for dealing with a completed block. This normally means displaying the completed block on the screen and recording it into the data queue; however, many other tasks are sometimes performed. These include:

- Handling calibrations, including split calibrations. (Split calibrations calibrate one microphone, and then the other. They are useful on a microphone like the B&K 4184, which uses acoustic calibrators — the calibration of one can throw off the other.)

Split calibration is handled by performing two separate calibration blocks, which get merged by the record routine. Also, a calibration includes a hidden "background check" block, which is used to verify that the difference between the background sounds and the calibrator is enough so that the calibration block is valid. When the calibration block is complete, the internal record of the calibration constants of each microphone is updated.

- Testing each block against the filters, and marking the block as important, unimportant, or rejected. A rejected block is displayed, but not recorded in the queues. (The actual filter code is contained elsewhere; record simply examines the results.)
- Translating "Block" structure into the data structure used by DATA.CPP, and calling the functions to add the block into the data queue.

Essentially, this procedure follows these steps:

- Initializes the block to be stored in the data queue to "not rejected" and "not important"
- If this is a calibration block and not a calibration background check
  - If this is the first block of a split calibration, hold the results for later
  - Otherwise:
    - If this is the second block of a split calibration, merge in the previous results
    - Determine if this is a valid calibration block using the background check results
- If this is a calibration block that was not rejected above
  - If the microphone is not a B&K 4184, update the microphone calibration constants for the calibrated microphones
  - Set the type for the block for the queue
  - Reset the recalibration interval to 2 minutes
  - Schedule the next automatic calibration
- Otherwise, if the block was rejected:
  - Reschedule the calibration for now plus the recalibration interval
  - Double the recalibration interval, but limit it to 2 hours
- Otherwise, if this is a manual, timed, or threshold block:
  - Set the type of the block for the queue
  - Assume that the block is important
  - Check if the block passes the filter

  - Using the filter, windy, and blast detection flags, determine if the block is important and/or should be rejected.
  - Copy in the acoustic data, display and store the block.
- Otherwise, if this is a background check block:
  - Display and hold the values for future use

Block::stop(tDSP *DSP)

Stops data collection for a block. Records the block, then clears it.

tDSP::tDSP(PROC_ID *proc)

Initializes the DSP data collection module variables. This includes clearing the manual and threshold data blocks, temporarily disabling threshold data collection until the monitor warms up, and resetting timeouts.

void tDSP::enableThresh()

Enables threshold data collection (if it is enabled in the configuration).

void tDSP::disableThresh()

Temporarily disables threshold data collection.

int tDSP::thrMode()

Returns 1 if data collection is enabled, 0 otherwise.

int tDSP::calibInProcess()

Returns 1 if a microphone calibration is in progress.

int tDSP::checkCalib()

Returns 1 if a calibration is due, 0 otherwise.

void tDSP::setFutureDB(struct *sdate, struct *stime, int len)

setFutureDB sets up the collection of a future manual data block. If len is 0, a pending future data block is cancelled. If len is not 0, any pending future data block is replaced with the specification provided.

The time for the future data block is set by sdate and stime. The term sdate contains the fields da_day, da_year, and da_mon (day, year, and month), and stime contains the fields ti_hour, ti_min, and ti_sec. (It also contains ti_hund, which is ignored.) At the

specified time and date, a data block of len seconds will be taken. Negative lengths mean that the calibration block should be turned on for the data block.

int tDSP::checkFutureDB()

This checks to see if a future data block has come due. If so, it returns the length of the data block that should be started in seconds (negative, if the calibrator should be enabled); otherwise, it returns 0.

enum Thresholds tDSP::checkThresh()

This function should check the the various thresholds against the trigger conditions set by the user in the configuration file, and then return which of the many different thresholds were exceeded. However, it works much better to not trigger unless all of the trigger conditions are satisfied, so the behavior of this function has been modified significantly.

The behavior of the threshold function is controlled by two #defines, CROSS_PEAK and THRESH_OR. Normally, CROSS_PEAK is enabled and THRESH_OR is disabled.

First, this function takes all of the relevant data from the “sam” structure in the tDSP class, which contains the latest data collected from the DSP board. These data are translated into acoustic decibel levels based on the latest calibration block taken by the field unit, and held to compare against the threshold. The actual peak measured by the microphone, a number in the range of 1-32767, is stored for comparison against the “absolute peak” threshold. If the monitor is set to use the input from both microphones, the cross sums are used. If the “CROSS_PEAK” option is set, the cross peaks are also used; otherwise, the geometric average of the two peaks is used.

If the “THRESH_OR” #define is included, all of the threshold conditions must be satisfied before a checkThresh() will return a value other than None. (If all of the conditions are satisfied, it arbitrarily returns a value of AbsoluteThresh.) However, if “THRESH_OR” is not defined, the conditions are checked in order (SELFlat, SELCWeight, PeakFlat, PeakCWeight, then AbsolutePeak), and the first condition that is satisfied will be returned.

Older versions of the field unit controller code had a problem that prevented the absolute threshold from working properly if the THRESH_OR mode was enabled.

int tDSP::checkFilter(void)

This works in much the same way as checkThresh, only it checks an accumulated block against the conditions, not a single sample. First, the variables for the various calibrated values are checked, and then they are compared against the threshold values for the filter. However, the filter always uses an AND structure — all of the filter constraints need to be satisfied for the filter to pass a block. The filter also checks to make sure that the sample length is in a particular range — this can be used to reject blocks in which wind constantly retriggers the field unit.

This function returns a 1 if the block passes the filter, or a 0 if the block does not pass the filter. The exact result of passing or failing the filter is controlled by the tDSP::record() function.

void tDSP::setcalconst(char chan, float lvl)

This function resets the calibration constant for a given microphone. (The calibration constant is the number that is added to the levels where 0dB = the minimum detectable noise level for the microphone.) It also updates the scale for the level bars at the bottom of the attached screen. (This is irrelevant for the modem console, as these bars are not present.)

void tDSP:setPost(unsigned char post)

This updates the cfg.posttrig_blocks configuration variable and has been kept only for backward compatibility.

void tDSP:setPre(unsigned char pre)

This updates the cfg.pretrig_blocks configuration variable and has been retained for backward compatibility.

void tDSP::synch(void)

This function synchronizes the call processor with the DSPs by reading the next address to be written from the DSP's synchronization registers. This is used to determine if new data are available in the tDSP::sample () method.

float tDSP::GetCurPeak(void)

This function returns the last flat-weight C peak (acoustic decibel) returned by the DSP. It is used for the realtime peak display on the console.

char tDSP::sample(void)

This function checks for new tenth-second blocks from the DSP; if a block is available, it retrieves the block and processes it.

To do this, it first checks to see if a new sample is available by comparing the DSP synchronization address (which contains the next address to be written) with the stored value for the address. If they do not match, it means a new data point has been written since the last time tDSP::sample() was called, and it needs to be collected.

If a new sample is available, sample() first calls Read_DPRAM_ Words_32() to collect one new sample, and advances the "current" pointer so it points to the beginning of the block that was just read. The sample read from the dual port RAM is read into the "sam" structure that is part of the tDSP class. Then the sam structure is adjusted to prevent bad data from causing domain errors for the log function. To do this, the absolute value of each value returned by the DSP is taken (it is possible for the cross peak and cross sums to be negative), and 10-37 is added to each returned value. This assures that, when the logarithm of the sample data is taken, a domain error will not result. (The logarithm is taken as part of the conversion to decibels.)

After the block is taken from the DSPs, it is processed. The first step in processing is to record the block into an active manual data block (the manual data block is also used for taking calibration blocks and calibration background checks; to tDSP::sample(), it is really just a timed sample). The new sums are added into the channel and cross sum fields. The peaks are compared against the highest peak already recorded and are updated if new data are available. The updated peak position is calculated using the peak position inside the sample (which is measured in microblocks of 24 samples — each tenth-second block has 200 microblocks) plus 200 times the length of the existing part of the block. The BDA blips (blast detection algorithm "hits") is then accumulated and the "windy" bit is set if applicable. (The "windy" bit indicates that any

part of a sample exceeded the wind threshold.) The variable "manual.downCnt" is decremented; when this variable reaches 0, the manual block is ended using the record() and clear() methods. The manual block can also be ended explicitly by clearing the "manual.sampling" variable.

After the manual/timed data blocks are dealt with, tDSP::sample() updates the displayed data calibration offset. Ideally, this would be done elsewhere; however, it works here, and no strong reason exists to move it. If the cfg.use_cal variable is set, the acoustic decibel level is shown in the LAST PEAK and previous block fields on the console; if it is left unset, all decibel levels are in relation to the minimum detectable sound. (The -96.33 dB correction is because all values returned by the DSP are multiplied by a factor of 65,536.) Using the calibration offset, the peak from this tenth-second block is saved in "lastpeak" for display.

The next step is to handle the threshold blocks. First "checkThresh" is called to see if the threshold is exceeded. The threshold is checked to see if it is disabled by looking at cfg.thrsam (the user threshold enable bit) and thr_disable (used to disable threshold blocks when the a calibration block is being taken.) If the threshold is disabled, threshold blocks are explicitly disabled.

If the threshold has been exceeded, a threshold block is started. This is done by storing the start time and date in the block using gettime and getddate, setting threshold.sampling, and then adding in the pretrigger information. The pretrigger information is retrieved from the circular data queue in the DSP dual-port RAM by retrieving previous blocks. This limits the number of pretrigger data blocks available to about 45, leaving some buffer space against delays elsewhere. No explicit check is made to ensure that this limit is not exceeded. The previous tenth-second blocks are not checked to determine if they were windy; however, the windy bit "sticks" for a certain amount of time.) The code then loops through the pretrigger blocks to add them to the threshold block. Finally, if a threshold block is currently being sampled, the block is updated with the current data stored in "sam." Like the manual block, the end of the threshold block is controlled by the downCnt variable. If the threshold is exceeded, the downCnt variable is reset to the posttrigger blocks configuration variable; otherwise, it is decremented. (The first tenth-second block in a threshold block

will always be the trigger block, so downCnt will be set.) When downCnt hits 0, or a programmed sample timeout is exceeded, the block is stopped and recorded using the threshold.record() and threshold.clear() functions.

Last, a word is returned telling the caller what is being done by returning the BDAblips, threshold.sampling, and manual.sampling flags.

tCommand::tCommand(tDSP *pDSP)

This initializes a command queue for the DSP class pointed to by pDSP.

The command queue class sequences DSP operations. It is used primarily to handle calibrations, which are a multistep process. It is also used for some delayed operations (for instance, if a manual block cannot be started immediately, it is placed in the queue). It uses a heap-ordered priority queue, where the next operation to be done is always stored at the top of the queue.

int tCommand::enq(Instr comm, time_t exTime, int argt)

This function inserts the value “comm” into the priority queue at time “exTime,” and resorts the queue so the first entry in the queue is the next instruction to be executed. “comm” is an enum that contains one of several valid commands:

TERMINATE: Ends the field unit control program

START: Start a manual sample block

TSTART: Start a timed data block. Argument “argt” is the length of the data block; a negative number indicates that the calibrator should be turned on for the data block.

CSTART: Start a calibration data block. The “argt” indicates which channels should be calibrated; 0 means use the default; 1 means channel 1 (the top microphone), 2 is the bottom microphone, and 3 is both. This simply fills the queue with a sequence of other commands (TSUSPEND, CBEGIN, CENABLE, CDISABLE, CEND) to implement the calibration procedure. The calibration procedure is described by the enq() commands contained in the implementation of the CSTART command. It also sets the “calibrating” flag.

CEND: This clears the “calibrating” flag. (DSP.cal_in_progress)

CBEGIN: This starts an actual calibration or background check sample. If the argt is positive, the indicated microphone (or microphones) is calibrated; the calibrator is assumed to be on. If argt is negative, a background check is started.

CENABLE: This enables the microphone calibrators. “argt” indicates which microphone calibrator is to be turned on: 1, 2, or 3 for both.

CDISABLE: This disables one or both microphone calibrators.

TSUSPEND: This temporarily suspends threshold data collection by setting DSP.thr_disable.

TENABLE: This reenables threshold data collection by clearing DSP.thr_disable. (Threshold mode cannot be enabled if the calibrators are on.)

STOP: This stops a currently running manual sample.

int tCommand:hot(void)

This returns 1 if the top command in the command queue is due.

int tCommand::print(void)

This displays all of the commands on the command queue. It is used only for debugging.

void tCommand::pop()

This removes the top entry in the queue, and moves the next earliest command into the first position of the command queue.

void tCommand::swap(int i, int j)

This swaps two entries in the command queue. It is used to maintain the heap ordering of the queue.

int tCommand:execute(void)

This function checks if the top command in the command queue is due; if it is, it executes it.

DSPERR.CPP and DSPERR.H

This source file contains only one function:

void printError(unsigned int error)

printError displays an error message associated with a DSP API (C4XAPP) error code.

ENVIRON.CPP and ENVIRON.H

ENVIRON.CPP contains the code that handles closed-loop environmental control. It reads out the temperature from the sensor on the WDT-501P watchdog board, and uses that information to control the heater and fan inside the case of the monitor unit.

unsigned char env_status_byte(void)

This function generates a status byte similar to the status byte for the old field unit design to report back to the base station. It indicates the temperature of the monitor unit to within 10-degree-C ranges, as well as over-voltage and under-voltage flags. It retrieves the current temperature from the value stored by env_periodic.

void env_periodic(void)

This function is called every iteration of the main loop. It has two main tasks: determine the current temperature inside the monitor case, and to try to keep that temperature within “reasonable” values.

To determine the temperature inside the case, it takes a reading from the watchdog board at most every clock tick. Eighteen readings are accumulated and averaged; the average temperature is stored in env_temp approximately once every second. (This smoothes out variation and appears to give a more precise measurement of the internal temperature.)

Also, every time through the loop, the current temperature is compared against a set of thresholds for which the field unit will take action. If the temperature is above 30 °C and the fan is not on; it is turned on; if the temperature is below 20 °C and the fan is on, it is turned off. If the temperature is below 12 °C and the heater is off, it is turned on; if the temperature is above 20 °C and the heater is on, it is turned off. (Note that these thresholds are somewhat lower than those of the original field unit.)

Last, this function generates environmental warning blocks. Temperature warning blocks are generated when the temperature inside the unit exceeds 45 °C or falls below 5 °C. At these extremes, damage to the unit may occur. After a warning block is generated, a flag is set to prevent another warning block from

being generated until the temperature falls below 43 °C or rises above 7 °C. This prevents memory from being filled with environmental warning blocks. Similarly, under-voltage and over-voltage warning blocks are generated whenever the voltage monitors inside the indicate the voltages are outside of tolerance — a new warning block will only be generated if the voltages return to a normal range.

void env_make_status(char type)

This function makes a status block out of the current environmental data. It must be passed an 'E' (temperature warning block) or 'F' (voltage warning block). After the block is generated, it is automatically displayed and stored.

int env_heat(int cntrl)

This function accepts the following options for cntrl:

1 Turn heater on
0 Do nothing
-1 Turn heater off.

It returns the current state of the heater: 1 is on, 0 is off. It will only turn the heater on or off if it has been more than 20 seconds since the last time the heater changed state — this allows testing and reduces oscillations.

int env_fan(int cntrl)

This function controls the fan in exactly the same was as env_heat controls the heater.

float env_get_temp(void)

This function returns the last temperature calculated by env_periodic().

int env_get_overvolt()

This function returns 1 if the watchdog board reports any voltages being too high.

int env_get_undervolt()

This function returns 1 if the watchdog board reports any voltages being too low.

MAIN.CPP and MAIN.H

MAIN.CPP contains the main loop and dispatches the various other functions. It also contains the startup and shutdown code for the field unit.

The top of MAIN.CPP contains a bunch of #defines for addresses of various components and other options. These will not need to be changed assuming the hardware is built with the same configuration as the test unit. These options are:

VIDEOBUFBASE:	0xB8000001 is the correct number for a VGA controller. A monochrome controller would require 0xB0000001.
WDT_ADDR:	0x2A0 is the base address for the WDT-501P watchdog board.
WDT_IRQ:	The WDT-501P watchdog board uses IRQ 10.
PPORT_BASE:	The "standard" first parallel port is 0x378; however, 0x3BC might be used also.
MIDNIGHT_REBOOT:	If this option is set, the field unit reboots each night at approximately midnight.

MAIN.CPP also declares a few global variables that are used in other modules. The variables "quit" and "rc" are used in CMD.CPP as part of the EXIT and KERMIT commands — if "quit" is set to 1, the monitor exits with the return code "rc." The variable "sh" provides the screen height to other modules, and the variables *dspPtr and *qPtr provide a pointer to the DSP class.

On startup, the noise monitor runs main(), the standard startup function for C and C++ programs. The first thing that the control software does upon startup is parse the command line options. These options point to the location of the DSP images to be loaded into the DSP RAM. Also, a few debugging options are provided. These debug options must follow the DSP paths. They are "/NOCAL" which disables automatic calibration, "/NOWM" disables the wind meter, and "/NOWD" disables the watchdog timer. Use of these three options makes debugging considerably easier, because the software will not crash or reboot if you break into it with a debugger. (If more than one of the "/NO" options is used, they must appear in the order described above.)

The control software then resets the parallel port, which turns off its outputs. (The parallel port outputs control the heater, fan, calibrators, and the status LED.) The screen is cleared, the banner is displayed, and then the field unit begins initializing itself. First the DSPs are initialized, and then the sampling class (tDSP) is initialized. ReadConfig() and ReadBuffers() are called, which

read in the field unit configuration and the data queue, and the number of reboots is incremented. (The field unit keeps track of the number of times that it is booted as a diagnostic.) Then the help system is initialized by calling init_help(). The parameter for init_help is the directory to look into for the help file. It is passed an empty string, which means the current directory (for the field unit, this is "A:\"). Ideally, this would read an environmental variable so that the help files could be changed without rewriting the ROM.

Next, the field unit initializes the serial port. Of the two serial ports (the console serial port and the modem port), the field unit initializes the modem port first. Initializing a serial port involves several calls. First, it calls init_port, which specifies the port (1 or 2), port speed, parity (always none), data bits (8), and stop bits (1). It then calls init_handl() and init_buffer_out to initialize the queued serial I/O driver. Once the modem port has been initialized, the console port is initialized and activated. The scmask() variable accepts a bitmap specifying a "1" in bit 0 for COM1 or bit 1 for COM2 — this mask indicates which serial ports receive console activity.

After the serial port drivers are initialized, the monitor sends a string to the console serial port to identify if it was mistakenly set to a port with a modem or another device that loops back incoming data. (This would result in massive amounts of garbage filling the input buffer.) To check this, it sends a string that includes 10 non-ASCII characters (character 255, which prints as a space). The input loop looks for these characters and disables the console port if they are found. The modem is then initialized by calling m_init_modem(). This function sends the modem initialization string to the modem, to initialize auto-answer and other modem options.

Next, the control program initializes the watchdog board by calling WDT_SetAddress(), and then initializes the wind meter (which uses the watchdog board to collect data) by calling wind_init, then calling wind_set_params and wind_set_wind meter to set the wind meter parameters and type as specified by the field unit's configuration. Last, the field unit enables the watchdog timer and hooks the watchdog card's interrupt (which is used to detect change-of-state on the wind meter input.) The watchdog card is set to reboot the system if it is inactive for 1 second by using a timeout of 1000 ms.

The last part of initialization is to display a blank input line and schedule an automatic calibration. The field unit automatically schedules a calibration for 3 minutes after startup, which allows plenty of time for the microphones to warm up.

Before the main loop begins, the monitor calls DSP.synch(). This synchronizes the control software with the DSPs, and ensures that the sample data that is taken represent the latest data available. A minimal amount of time should pass between the call to DSP.synch() and the first call to DSP.sample(), which will retrieve any new samples that are available.

After the initialization section of MAIN.CPP comes the main loop, which dispatches all of the important tasks of the monitor. The first part of the main loop is the basics: handling incoming data blocks, queued events, incoming serial communications, status, handle any console keystrokes, and closed-loop environmental controls. The system handles these basics by calling single functions located elsewhere — updatelites, checkSerial(), and checkConsole() are in MAIN.CPP; env_periodic is in ENVIRON.CPP; and DSP.sample() is in DSP.CPP. After the basics are complete, a few more select tasks need to be done.

First, 1 out of 30 times through the loop (which helps ensure that the controller does not fall behind while under heavy load), the field unit writes a single data block out to disk. It does this by calling WriteConfig(), which will return 0 if it has nothing to write. If WriteConfig() has nothing to do, WriteSegment() is called to write a segment of the data queue to disk.

Next the field unit checks to see if a calibration is pending using DSP.checkCalib(); if so, it enqueues a configuration request (CSTART) onto the DSP event queue. The DSP then checks for a future data block. If a datablock is waiting, it is enqueued. If none is pending, DSP.checkFutureDB returns 0.

Next, the controller checks to see if any changes to the serial port configurations have been made. If so, and no one is using the modem console, the serial port subsystem is restarted. This allows changes to the serial configuration to be made from the serial port or remotely. Without the check for a remote user, one would immediately be knocked offline by any changes.

Finally, if the MIDNIGHT_REBOOT compile-time option is set, the field unit checks for the BIOS clock rollover. (If the BIOS clock rolls over, it indicates that midnight has passed.) If so, it explicitly disconnects the field unit from the base station, and then reboots if not currently connected. (This gives the field unit time to cleanly disconnect from the base station, and prevents lost or duplicated data blocks.)

The last part of main() is the shutdown code. After cleaning up the console somewhat, the shutdown proceeds. First it creates an “exit timeout” variable to protect against the code that writes data out to disk from not terminating the

loop. The field unit will only prompt the watchdog while waiting for WriteConfig() and WriteSegment() to complete if it is less than 60 seconds after flushing of the queues starts. After setting the timeout, the field unit flushes the queues by executing WriteConfig() until it returns 0 (if the configuration needs to be synchronized, WriteConfig() may need to be called twice to update both copies). It then calls WriteSegment() until it returns 0 as well. (WriteSegment() may need to write as many as 64 blocks to disk.) After synchronizing the data queues, the serial console is turned off, the DSPs and the DSP control library closed down, and the watchdog reprogrammed for 30 seconds. These steps are to ensure that, should KERMIT or the watchdog monitor TSR fail to start, the field unit will return to the data collection mode. Finally, exit(rc) is called to terminate the program with return code rc. The return code is used to determine whether KERMIT should be started, or the field unit should simply reboot.

The field unit can return an error code of 1 to 100 to indicate internal errors. If this happens, the field unit will automatically fall back into the ROM version of the program code. Error code 0 indicates normal completion; the field unit will reboot and start again. Error code 101 indicates that WDMTSR should be started on COM1, and then KERMIT should be executed; 102 indicates WDMTSR is to be started on COM2.

MAIN.CPP also includes several other routines that implement various portions of the main loop. These include:

void SpinnyThing(void)

> This function updates a “spinny thing” in the corner of the console that indicates that the field unit’s main loop is executing. It also prompts the watchdog, which postpones the automatic reboot for another second.

void BlinkLED(int ontime, int offtime)

> This function blinks the Status LED (the yellow LED on the 24-V board) on and off. The ontime and offtime numbers are given in clock ticks — the LED is on if { (ticks-since-startup%(ontime+ offtime) < ontime }. It should be called as frequently as possible to ensure that the LED has the correct duty cycle.

void checkSerial(void)

> This function calls m_periodic() to handle the low-level modem protocol (which includes placing and receiving calls, and sending and receiving packets). It then checks to see if a packet or an

acknowledgment of a sent packet has arrived; if so, it calls process_pkt() or process_ack() to handle the packet.

Last, it checks to see if any important blocks have been collected, or if the number of blocks in the queue is greater than the threshold number of blocks before it calls the base station. If either of these cases is true, it calls m_connect() to attempt to call the base station. (m_connect will ignore any requests that have not changed from previous requests.)

void checkErr(UINT error)

This function checks for failed DSP initialization functions. It then displays any applicable error code ("No Error" is displayed if no errors have occurred), and exits with exit code 1 if an error has occurred.

void createbars()

This function draws the scale for the bars shown on the bottom of an attached monitor. It is called after calibrations or if the "SET USE_CAL" function is invoked.

void bar(int row, int start, int endbar)

This function fills in a bar on the bottom of the screen with red from column "start" to column "endbar," and with black the rest of the way to the right edge of the screen.

void updatebars(float CSEL, float FPK)

This function is a stub, whenever a block is taken, that simply records the values passed into it into a more accessable location: status_sel and status_peak global variables.

void updatelites(char sam, tDSP *dsp)

This function updates all of the status indicators on the main screen, and calls BlinkLED() and SpinnyThing(). The status bar at the top of the screen is drawn using scslprintf(), which echoes the status bar to the serial and modem consoles. The rest of the status indications are only displayed on the attached VGA monitor.

The status bar is explained by the help entry "HELP STATUS." Because some terminal programs have difficulty with the ANSI sequences used to create the status bar, it can be disabled. If the status bar is disabled (with STATUS OFF), the status bar is not

displayed on the main screen either. However, all of the bars and lights on the bottom of the screen are still displayed and updated.

The pokeb(0xb800, ...) variables are used to update attributes and characters shown on the screen. 0xb800 is the segment of a VGA video controller — many of the bars and lights will fail on a monochrome video adapter.

char display_input(char *string, int line)

This function displays the last 79 characters of the user's input line and the cursor on the attached VGA display. This function will not fail on a monochrome adapter if the VIDEOBUFBASE variable is changed. The string to be displayed is *string (if it is longer than 78 characters, it scrolls off to the left), and line is the line number to display the string on. A blinking block character is displayed at the end of the string as a cursor.

char checkConsole(void)

This function checks for any characters waiting from the keyboard or a serial console. If characters are waiting, it adds the characters to the input string if they are printable and the maximum string length of 127 has not been exceeded. If the character is 13 (carriage return), it calls do_command() to execute it. If the character is 8 (BS) or 127 (DEL), it erases the last character from the input command; 18 (CTRL-R) resends the string to any remote serial consoles, and 27 (ESC) cancels any waiting string and erases the command line.

This function also checks for any characters over ASCII 127 to appear in the first few seconds of operation. If they appear, it is assumed that there is echo from the console serial port, and scmask(0) is called to disable it.

void WDThook(int status)

This function is called whenever a watchdog interrupt occurs. If the watchdog interrupt source is a 0 to 1 transition of the wind meter input, the wind_blip() function is called. The wind_blip() function maintains counters of how many wind meter 0 to 1 transitions have occurred; these are used to measure wind speed.

PACKET.CPP and PACKET.H

These functions handle the high-level packet interface and would need to be expanded to add a more complete data interface to the field unit (i.e., to allow collection of all the data that the monitor collects, not just parts of it). At the lowest level, the protocol used by the noise monitor is a basic command/response scheme. The monitor receives a request from the base station and sends back either a response packet or an acknowledgment. If the response is a packet, the monitor waits for an acknowledgment before it assumes that the packet has arrived. The field unit follows the Lendrum/Averbuch protocol specification, attached as Appendix F.

PACKET.CPP includes the following functions:

int process_ack(void)

This function processes an incoming acknowledgment, if one exists. If not, it returns a 1. As part of processing the acknowledgment, the field unit will delete any data queue block that has been sent to the base station and acknowledged. (The block is not deleted until the acknowledgment is received. Therefore, it is possible for the block to be sent twice if the acknowledgment is lost and then the field unit disconnects; however, it is unlikely for the block to be lost.)

int process_pkt(void)

This function accepts an incoming packet and dispatches it based on its type (which is identified by the first character in each packet.) The packet is dispatched to the function pkt_inX(*packet*), where X is the type, and *packet* is a null-terminated string containing the packet.

void Amakeblock(struct DataBlock *datum, char *response)

This function converts the data block *datum from the internal format into the format specified by the Lendrum/Averbuch protocol specification. The converted packet is returned in the area pointed to by response.

void pkt_inA(char *packet)

This function checks for any packets waiting to be sent. If a packet is waiting, the function calls Amakeblock to construct a block from the oldest data block in the queue, marks that a data

block is currently in transit, then calls m_putpacket to send the block. (The datablock_pending flag indicates that, when an acknowledgment is received, the oldest block in the data queue is to be deleted.) If no blocks are waiting to be sent, a block of type "C" is constructed and sent instead. This block indicates that the data queues are empty.

int Bmodeget(int mode, char *response)

This function returns, in the memory pointed to by *response*, a packet giving the current state of the mode specified by *mode*. It does not handle a *mode* of zero, which specifies that all values should be returned. The exact format of the returned packet is documented in the Lendrum/Averbuch specification. The returned packet includes only the data itself; the type header is added by the calling procedure. If *response* does not point to an empty null-terminated string, the mode string constructed is appended to the existing string.

int Bmodeset(int mode, char *set)

Bmodeset accepts a mode parameter in *mode*, and sets it to the data in *set*, as described by the Lendrum/Averbuch specification. If mode is 14 or 10 (the specification and the field unit software disagree on this), the field unit empties its data queues. The received mode is placed into the cfg structure, and then ConfigDirty() is called to request flushing the configuration to nonvolatile storage.

void pkt_inB(char *packet)

pkt_inB handles type B packets, which set and retrieve field unit modes. If mode (the second character of the packet - 0x20), is 0x1A, a calibration request is queued. If mode is between 0x10 and 0x1F, mode-10 is passed to the Bmodeset function with the reset of the packet, which sets a field unit mode. If mode is between 0x01 and 0x0F, it is passed to Bmodeget. If mode is 0x00, Bmodeget is called repeatedly to accumulate all of the mode information into a single packet. Once the packet to send is assembled, it is sent.

Note that, if a mode is set, it is then retrieved and sent back as a confirmation.

void pkt_inC(char *packet)

This function simply assembles a packet type 'G' with the unit's number and sends it.

void pkt_inD(char *packet)

The listen mode is not supported by this field unit design, so the field unit simply acknowledges the listen request and hangs up. This should be corrected in a future hardware revision.

void pkt_inE(char *packet)

This function adds a TSTART event to the DSP's command queue, scheduling an immediate, manual data block. It then acknowledges the receipt of the manual data block request.

void pkt_inF(char *packet)

This function schedules a future data block. Note that only one future data block can be scheduled. If a new future data block is scheduled before a previous block is taken, the previous block specification will be overwritten. Because setFutureDB echoes the data in the block, the display is modified in strange ways — extra code is required to clean up the console.

Once the future data block is scheduled, an acknowledgment is sent back to the base station.

PPORT.H

This header provides macros to manipulate the peripherals attached to the parallel port. These peripherals are:

CALIB1:	The calibrator on microphone #1, the top microphone
CALIB2:	The calibrator on microphone #2, the bottom microphone
LED:	The status LED attached to the 24-V board
FAN:	The case exhaust fan
HEATER:	The 1000-W heater/fan attached to the left side of the field unit.

The macros provided by the file are:

PP_RESET:

PP_RESET turns off all the peripherals attached to the parallel port.

PP_SET(*x*):

PP_SET turns on the peripheral *x*, where *x* is one of the peripheral names above.

PP_CLEAR(*x*):

PP_CLEAR turns off the peripheral *x*.

PP_GET(*x*)

PP_GET returns the current state of peripheral *x*, where 1 is on and 0 is off.

PROTOCOL.CPP and PROTOCOL.H

These source files contain the code to implement the low-level data protocol described by the Lendrum/Averbuch specification. This protocol is compatible with (though not exactly identical to, especially in terms of timing) the original CERL field units, and fully compatible with the base station software when running at 300 bps using transmit pacing.

This module is a little hard to follow because of the myriad of different modes supported by the protocol implementation. The closest compliance to the Lendrum/Averbuch specification is achieved by setting the protocol to do transmit pacing, which limits the number of output characters sent to approximately the bps rate of the outgoing modem connection. Without transmit pacing, the field unit's automatic retransmissions of packets can overrun a 300 bps connection, and the base station expects nearly instant responses to acknowledgement (ACK) or negative acknowledgement (NAK) packets. The transmit pacing reduces the overrun, allowing proper behavior. For practical purposes, this module can (and should) be considered a black-box implementation of the low-level protocol. All of the functions here return immediately; any future actions are handled by the m_periodic function.

This function contains the following user functions:

void m_periodic(void)

This should be called from the field unit's main loop. It handles protocol maintenance, transmit and receive pacing, packet retransmissions, dial outs, and most of the protocol.

int m_console_off(void)

This function disables the field unit modem console mode. This automatically occurs when the modem's carrier is lost.

void m_init_modem(void)

This function requests that the modem be initialized by resending its initialization string.

int m_connected(void)

This function returns 1 if the modem is connected to a base station. (Note that it returns 0 if the modem is connected but in console mode.)

void m_connect(char *phone)

This function requests a connection be made to the specified phone number. Note that subsequent requests to connect with the same phone number will not have any effect; additional attempts will be made at intervals that begin at 2 minutes and double every time thereafter. The maximum interval is 2 hours.

Calling m_connect() with a null string as the **phone* parameter will cancel any pending redial attempts.

void m_disconnect(void)

This function disconnects the modem from the base station by sending multiple data link escape (DLE) characters. The modem will then be forcibly disconnected after a half-second delay.

unsigned int m_get_speed(void)

This function returns the current speed of the connection to the base station.

int m_getpacket(char *pkt)

This function returns a waiting packet, if available. The packet goes into *pkt, and the return code is one of the following:

0	No pending packets
>0	Length of packet received and placed into *pkt
1	An ACK packet was received from the base station.

int m_putpacket(char *pkt)

This function sends a packet. It returns 0 if a packet is still pending (has not been acknowledged); otherwise, it returns 1. There is currently no way of canceling an outgoing packet that has not yet been acknowledged. (A packet can be acknowledged by either the receipt of an ACK character or the receipt of a valid packet.)

int m_acknowledge(void)

This function sends an acknowledgment (an ACK character) to the base station. It returns a 0 if another packet is currently pending; otherwise, it queues the acknowledgment for transmission.

int m_inpacket(void)

This function has the same return codes as m_getpacket, but does not receive an incoming packet. It is intended as a check for waiting packets.

int m_outpacket(void)

This function returns the length of any pending output blocks. The latest version of the monitor code changes this function so it returns a -1 if an acknowledgment is pending. Older versions returned 0 if an acknowledgment was pending, which is a “bug.” If no output blocks are pending, this function returns a 0.

The following additional functions are provided by the PROTOCOL.CPP module, but there should be no need for code outside PROTOCOL.CPP to use them.

int m_console_esc(void)

This function returns 1 if a console escape has occurred. There should be no need to use this function from a user program.

int m_console_on(void)

This function enables the field unit modem console mode. Normally, the field unit console is enabled and disabled automatically by the protocol module, so there should be no need for this function to be used in user code.

The interface to PROTOCOL.CPP hides quite a bit of its complexity. It has several variables and functions that warrant more detailed descriptions, and a few private functions that are not described above. The meanings of the private

variables are all documented in the code's comments; however, a few of the functions are difficult to follow.

Local functions in PROTOCOL.CPP include:

void m_dump_output(void)

This function clears any pending output in the data buffers. Unfortunately, it does not work efficiently, because the Sportster 33.6 kbps modem will accept data into its buffers as fast as it can be transmitted by the machine, not at a flow rate limited by the "connection speed" programmed into its universal asynchronous receiver/transmitter (UART). Data that have already arrived in the modem's buffer cannot be canceled, so transmit pacing is used instead. This function also cancels any pending output block, which effectively cancels the paced transmissions.

void m_flowcontrol(int status)

This function is automatically called by the status change interrupt on the modem line. It is used for several purposes. First, it updates a line monitor on an attached video graphics array (VGA) screen (if the LINE_MON #define is set). Second, it is responsible for flow control (by disabling the transmit interrupt whenever Clear to Send [CTS] is cleared, and re-enabling it when CTS is set). If the modem hangs up, this function turns off the modem console, resets the flow control variables, and sets the speed of the modem to full speed. It also somewhat supports "braindead modems" enabled by the "BRAINDEAD_MODEM" compile time option, which is intended to help deal with modems that do not properly handle the data carrier detect (DCD) line until they are initialized. It is not certain that this option works properly, however. (This code was intended to allow an Octagon 2400-bps wide-temperature-range modem to work; however, this modem was found to have other problems that made it unsuitable for the field unit.)

This function also reinitializes the modem if a rising edge is seen on the data set ready (DSR) signal, which means that the modem has been turned off and back on or plugged in (this is not really applicable to the internal modem used in the field unit). Also, if the modem detects a ring, it cancels any pending outgoing calls, and records the time of the ring detection.

chksum(x) (MACRO)
This macro updates the current block checksum (stored in m_inblock_checksum) to include the additional character *x*.

record(x) (MACRO)
This macro adds a new character *x* to the incoming block m_inblock, and updates the length m_inblock_len.

dump() (MACRO)
This macro resets the length of the input block and the input state machine, and sends a NAK to the modem. It is used when something appears to be wrong with the incoming data to request another attempt to send the packet.

bcase (MACRO)
A shorthand for a break followed by another case statement.

void m_process_char(int chr)
This function is called whenever a new character is received from the modem. It handles all of the low-level details of the packet formation, as well as other things such as the console escapes. Most importantly, this includes the packet-receiver state machine.

SERCON.CPP and SERCON.H

These two source files provide the console interface to the rest of the program. All console accesses should go through one of these functions (scprintf, scputs, scputch, scslprintf, scgetch, scgetche, and scwaiting) to ensure that they work properly in the serial console mode and for remote access through the modem.

The functions defined in this module are:

int scmask(int port_mask)
This updates the port mask. If the input port_mask is –1, the current port mask is returned.

There are four valid port masks:

0 no serial ports in console mode
1 COM1 is a console, COM2 is not
2 COM2 is a console, COM1 is not
3 both COM1 and COM2 are consoles.

Normally, the console serial port will always be enabled. The modem port will be enabled by the software when the escape sequence of 10 Ctrl-A characters is received.

int scprintf(char *format, ...)

This is the same as Borland's cprintf() which puts (a printf() for Borland's "conio" text formatting functions, but sends results to serial console as well.

int scputs(char *str)

This is analogous to Borland's cputs(); or c's puts() function. It sends a string to all active consoles.

int scputch(int ch)

This is analogous to Borland's cputch(); c's putch(). It sends a single character to all active consoles.

int scgetch(void)

Like getch(), waits for a single character to become available from any active console, and returns it without echo.

int scgetche(void)

Like getche(), waits for a single character to become available from any active console, echoes it, and returns it.

int scwaiting(void)

Returns 1 if a character is waiting and getch() or getche() will return immediately if called; 0 otherwise. Since a "realtime" system is being used, where unterminated delays cause problems, always check for an available character with scwaiting() before actually retrieving a character.

int scslprintf(char *fmt, ...)

This formats a string as printf would, then displays it on the status line of all consoles. Note that the status line is not "protected" on the serial consoles, so this needs to be called on a regular basis. (The status line will only be updated if no data are pending, and sufficient time has elapsed since the last update.)

WIND.CPP and WIND.H

These two files contain the code to handle the wind meter, which is implemented using the isolated inputs on the watchdog card. These inputs trigger an interrupt, which is dispatched by the WDT501 handling code to wind_blip() function. For timing, the wind meter also hooks the BIOS 55-ms timer interrupt. The wind meter is assumed to be of the chopper type, and the number of pulses per second is proportional to wind speed.

Although the interface with the base station uses the number of ticks in a sample interval to measure wind speed, proper operation of the field unit's console commands require that the field unit be aware of the actual performance characteristics of the attached wind meter. The wind meter's response is specified with two numbers — the multiplier and the addend — which define the following relationship between the number of measured pulses per sample interval and the actual wind speed:

$$\frac{Pulses}{SampleInterval} = ((multiplier \cdot speed) - addend) \times SampleInterval$$

To mark blocks as windy, set a local "windy_flag." When this flag is set, "wind_countdown" is also set to be the number of remaining clock ticks until the windy flag times out. This is decremented every timer tick so that, when it reaches 0, the windy flag is cleared. If the threshold is exceeded before the countdown completes, the countdown is reset to its original value.

The wind meter reads its wind meter configuration from the configuration path (read from the configuration file). It is stored in a file called "WIND.TXT," which has the following format:

<windmeter name>%<addend>%<multiplicand>%

where the wind meter name is used to refer to the configuration when setting the wind meter by name.

Interface:

void wind_init(char *path)

This initializes all of the interrupt handlers necessary to deal with the wind meter. It hooks the clock interrupt (which also is used for timing much of the rest of the code) and the watchdog interrupt

vector, reads the wind meter table, and registers shutdown code to unhook the interrupts it hooks.

void wind_deinit(void)

This unhooks the clock interrupt, which allows the monitor to be shut down safely. Initialization adds this to the exit hook, so it will be run automatically upon termination.

void wind_set_params(int threshold, float interval, float timeout)

This sets the windy-flag parameters. The threshold is the number of wind meter pulses that must be received in a sample interval. The interval is the length of time that the wind meter code accumulates pulses. Both the interval and the timeout are measured in seconds. After the threshold is exceeded, the windy flag is set and remains set for timeout seconds. Note that timeout should be greater than or equal to the sample interval to ensure that the windy flag stays on for the entire sample interval.

void wind_get_params(int *threshold, float *interval, float *timeout)

This returns the previously set threshold, interval, and timeout.

void wind_set_wind meter(float multiplier, float addend)

This sets the multiplier and addend used to translate between absolute wind speed (in miles per hour) and the ticks returned by the spinning wind meter. These values are used as shown in the formula on page 126.

void wind_get_windmeter(float *multiplier, float *addend)

Returns the current values for the multiplier and addend.

void wind_set_wm_string(char *string)

This function looks up the wind meter named in “string” in the WIND.TXT file, and automatically sets the addend and multiplier based on those values.

char *wind_get_wm_name(void)

This looks for a wind meter in WIND.TXT that matches current parameters (addend and multiplier), and returns that name if it exists. It returns “CUSTOM” if no such wind meter is found.

float wind_speed(void)

This returns the current wind speed, in miles per hour. It uses the conversions set in the formula on page 126.

float wind_ticks_to_mph(int ticks)

This converts ticks per sample interval into miles per hour, and returns the result.

int wind_mph_to_ticks(float speed)

This calculates the number of ticks per sample interval resulting from the given wind speed, then rounds off the result to the nearest integer and returns it.

clock_t clock(void)

Returns the number of 55-ms clock ticks since the wind meter code was initialized. (This is a working version of the clock() function implemented by Borland C++.)

void wind_suspend(void)

This temporarily disables the wind meter IRQ, disabling collection of wind data. (Any attempts to determine wind speed when the wind meter is disabled will result in an output of 0 mph/0 ticks.)

void wind_enable(void)

This function reenables the wind meter IRQ, allowing the wind meter to work normally.

Internal functions:

void interrupt wind_timer_int(...)

This is the timer interrupt handler. It keeps track of all of the timing issues involved in the wind meter, setting or clearing the windy flag as warranted by the set timeout, threshold, and sample interval. It also maintains a count of the number of clock ticks since the wind meter was initialized, which is used for timing in other parts of the code. After it does all of its work, it calls the original timer interrupt vector to ensure proper operation of the DOS clock.

void wind_blip(void)

This function simply adds one to the count of the number of wind meter ticks seen. It is called by the watchdog interrupt code.

Field Unit Boot Code

CONFIG.SYS

```
[menu]
menuitem=monitor,Standard Noise Monitor operation
menuitem=dos,DOS Prompt for System Maintenance
menuitem=reflash,Automatic Reflash (Put image floppy in disk drive)
menuitem=killrom,Reset code and setup to ROM defaults
menudefault=monitor,3

[common]
device=dos\himem.sys
device=dos\pcfsrdvr.sys
device=dos\ramdrive.sys 2048 /e
dos=high
stacks=9,512

[monitor]
[dos]
[reflash]
[killrom]
```

AUTOEXEC.BAT

```
path c:\;c:\data;a:\;a:\dos
set dsp_path=a:
set temp=d:\
mkdir c:\data
if exist c:\startup.bat call c:\startup.bat
if %config%==monitor monitor %dsp_path%
if %config%==reflash reflash
if %config%==killrom killrom
```

MONITOR.BAT

```
:start
if exist c:\main.exe c:\main %dsp_path%\mondsp_1.out %dsp_path%\mondsp_2.out
if not exist c:\main.exe main %dsp_path%\mondsp_1.out %dsp_path%\mondsp_2.out
if errorlevel 102 goto com2
if errorlevel 101 goto com1
if errorlevel 1   goto romver
goto end
:romver
a:\main.exe a:\mondsp_1.out a:\mondsp_2.out
if errorlevel 102 goto com2
if errorlevel 101 goto com1
:error
Echo Noise monitor failed startup!
goto end
:com1
wdmtsr 1
goto kermit
:com2
wdmtsr 2
goto kermit
:kermit
kerlite
goto end
:end
reboot
```

Appendix D1: MON7_2 code (CPU_B) Blast Detection Algorithm

```
 1 ********************************************************************
 2 *                                                                  *
 3 *                                                                  *
 4 *   Date     : 11 JUL 96                                           *
 5 *   Title    : MON7_2.ASM                                          *
 6 *                                                                  *
 7 *   MON7_2 receives sampled data from C40 communication link 3&4.  *
 8 *   The output word is then transmitted back along comm port 3.    *
 9 *                                                                  *
10 *   This program is part of the prototype monitor program and      *
11 *   must be loaded on the second processor module.  MON7_1 must    *
12 *   be loaded on the first module.                                 *
13 ********************************************************************
14         .sect   “chahist”       ;channel a history storage area, length=128 at 0x00300000
15
16 CHA     .space 128
17
18
19         .sect   “chbhist”       ;channel b history storage area at 0x003000400
20
21 CHB     .space 128
22
23
24         .sect   “ehist”         ;energy history storage area, length=690 at 0x00300800
25
26 EH      .space 690
27
28
29         .data
30
31 STACK           .word   002ffc00H       ; Define stack space
32 IVECTAB         .word   00308000H       ; Interrupt vector table
33 CHAHIST         .word   CHA
34 CHBHIST         .word   CHB
35 ENHIST          .word   EH
36
37 COMM_PORT3_CTL  .word   00100070h       ;Port 3 control
38 COMM_PORT3_OTP  .word   00100072h       ;       output
39 COMM_PORT3_INP  .word   00100071h       ;       input (for channel a data)
40 COMM_PORT4_CTL  .word   00100080h       ;Port 4 control
41 COMM_PORT4_OTP  .word   00100082h       ;       output
42 COMM_PORT4_INP  .word   00100081h       ;       input (for channel b data)
43
44         .global MAIN                    ;assign global variables
45         .global CUREF
46         .global CHAF3R
47         .global CHAF3I
48         .global CHAF2R
49         .global CHAF2I
50         .global CHBF3R
51         .global CHBF3I
52         .global CHBF2R
53         .global CHBF2I
54         .global MCHAF2
55         .global MCHAF3
56         .global MCHBF2
57         .global MCHBF3
58         .global ZC
```

```
 59          .global XC
 60          .global ACA
 61          .global ACB
 62          .global DETFUN
 63          .global NEWSA
 64          .global NEWSB
 65          .global NEWCHA
 66          .global NEWCHB
 67          .global OLDCHAX
 68          .global OLDCHBX
 69          .global ICRDY3
 70          .global READYA
 71          .global READYB
 72          .global DFCHK
 73          .global DCHK
 74          .global STAGE1
 75          .global OUTSTAGE
 76          .global NOOUT
 77          .global DECAYOUT
 78          .global fpinv
 79          .global GET_DATA
 80          .global BACKLOG
 81          .global TEST
 82          .global TEST3
 83          .global TEST4
 84          .global BEGN_ALGR
 85          .global ENABLE
 86
 87 COS3    .set 0.99518             ;cos(2*pi*126/128)
 88 SIN3    .set -0.09802            ;sin(2*pi*126/128)
 89 COS2    .set 0.99880             ;cos(2*pi*127/128)
 90 SIN2    .set -0.049068           ;sin(2*pi*126/128)
 91 NN      .SET 0.1
 92 NNN     .set 0.01
 93 CONST   .SET 5.12
 94 CONST2  .SET 100.0
 95 ZCTHR   .SET 0.49                ;square of the correlation ratio threshold
 96 ALPHA   .set 0.99999
 97 ALPHAN  .set 0.998721            ;alpha^128
 98 CUREF   .float 0.0               ;current energy function value
 99 DETFUN  .float 0.0               ;DETERMINATION FUNCTION CALC FROM EF
100 CHAF3R  .float 0.0               ;real part of third point of fft chA
101 CHAF3I  .float 0.0               ;imaginary part of third point of fft chA
102 CHAF2R  .float 0.0
103 CHAF2I  .float 0.0
104 CHBF3R  .float 0.0
105 CHBF3I  .float 0.0
106 CHBF2R  .float 0.0
107 CHBF2I  .float 0.0
108 MCHAF3  .float 0.0               ;CHAF3R^2+CHAF3I^2 , “magnitude squared”
109 MCHAF2  .float 0.0
110 MCHBF3  .float 0.0
111 MCHBF2  .float 0.0
112 OLDCHA  .float 0.0               ;XA(n-N)
113 OLDCHB  .float 0.0               ;XB(n-N)
114 NEWSA   .word   0                ;new integer sample from port ch A
115 NEWSB   .word   0                ;new integer sample from port ch B
116 NEWCHA  .float 0.0               ;XA(n)   floating point
117 NEWCHB  .float 0.0               ;XB(n)   floating point
118 OLDCHAX .float 0.0
119 OLDCHBX .float 0.0
120 ZC      .float 0.0               ;PREVIOUS zeroth lag cross-correlation value
121 ACA     .float 0.0               ;AUTOCORR OF CHA
122 ACB     .float 0.0               ;AUTOCORR OF CHB
123 OLDEF   .float 0.0               ;oldest value of ef
124 CSAMP   .WORD   0                ;SAVE CURRENT SAMPLE
125 XC      .float 0.0               ;CROSSCORRELATION VALUE
126 OUTCHA  .float 0.0               ;CHA OUTPUT
127 READYA  .WORD   0                ;new cha input sample ready
128 READYB  .word   0                ;new chb input sample ready
129
130
131 IIEMASK .word   44000h           ;Interrupt enable mask - for ICRDY3 and ICRDY4
```

```
132
133
134          .text
135          ; Define the interrupt vector table...
136         BR      MAIN            ; Reset vector
137         .word   0h              ; NMI
138         .word   0H              ; TINT0
139         .word   0H              ; INT0
140         .word   0H              ; INT1
141         .word   0H              ; INT2
142         .word   0H              ; INT3
143
144         .space  18              ; Reserved space and unused port 0 and
145                                 ; 1 interrupts.
146         .word   0h              ; Input channel full, port 3
147         .word   ICRDY3          ; ICRDY3
148         .word   0h              ; OCRDY3
149         .word   0h              ; Output channel empty, port 3
150
151         .word   0h              ; Input channel full, port 4
152         .word   ICRDY4          ; ICRDY4
153         .word   0h              ; OCRDY4
154         .word   0h              ; Output channel empty, port 4
155
156                                 ; Start of program proper...
157
158 MAIN:   LDP     @STACK,DP       ; Initialize the stack
159         LDI     @STACK,SP
160
161         LDI     @IVECTAB,R0     ; Set up the interrupt vector table
162         LDPE    R0,IVTP
163
164
165 ************************************************************************
166 *  Setup comm_ports and zero the input FIFOs                            *
167 ************************************************************************
168
169         LDA     @COMM_PORT3_INP, AR0    ; Input FIFO addr channel a data
170         LDA     @COMM_PORT4_INP, AR1    ; Input FIFO addr channel b data
171         LDA     @COMM_PORT3_OTP, AR2    ; output FIFO addr recogn. data to proc 1
172
173         LDA     @COMM_PORT3_CTL, AR5    ; control registers
174         LDA     @COMM_PORT4_CTL, AR6
175
176         LDI     *AR5,R8                 ; get control register3
177         LDI     *AR6,R9                 ; get control register4
178         OR      00000008h,R8            ; change bit3 to 1 to halt input FIFO3
179         OR      00000008h,R9            ; change bit3 to 1 to halt input FIFO4
180         STI     R8,*AR5                 ; write change to control register3
181         STI     R9,*AR6                 ; write change to control register4
182
183
184 TEST3   LBU1    *AR6,R8                 ; load byte 1 of comm_port3 control reg.
185         LSH     4,R8                    ; shift left by 4 bits
186         LBU0    R8,R8                   ; load only 1st byte
187         LSH     -4,R8                   ; right shift by 4 bits
188                                         ; R8=number of words in INPUT FIFO3
189         LDI     0,R9
190         CMPI    R8,R9
191         BZ      TEST4                   ; go on and test FIFO4 , FIFO3 is empty
192         SUBI    1,R8                    ; compute # words-1
193         RPTS    R8                      ; repeat next instruction (#words in FIFO-1) times
194         LDI     *AR0,R10                ; load FIFO values to clear
195
196
197 TEST4   LBU1    *AR5,R8                 ; load byte 1 of comm_port3 control reg.
198         LSH     4,R8                    ; shift left by 4 bits
199         LBU0    R8,R8                   ; load only byte0
200         LSH     -4,R8                   ; right shift by 4 bits
201                                         ; R8=number of words in INPUT FIFO3
202         LDI     0,R9
203         CMPI    R8,R9
204         BZ      ENABLE                  ; go on and enable interrupts, FIFO's are empty
```

```
205         SUBI    1,R8                    ; compute # words-1
206         RPTS    R8                      ; repeat the next instruction (#words-1) times
207         LDI     *AR1,R10                ; load FIFO values to clear
208
209
210 ENABLE  OR      @IIEMASK, IIE           ; Enable ICRDY3 and ICRDY4 interrupts only
211         OR      2000h, ST               ;        CPU global int
212
213         LDI     *AR5,R8                 ; get control register3
214         LDI     *AR6,R9                 ; get control register4
215         XOR     00000008h,R8            ; change bit3 to 1 to unhalt input FIFO3
216         XOR     00000008h,R9            ; change bit3 to 1 to unhalt input FIFO4
217         STI     R8,*AR5                 ; write change to control register3
218         STI     R9,*AR6                 ; write change to control register4
219
220         LDI @CHAHIST,AR3        ;set pointer to chA data
221         LDI @CHBHIST,AR4        ;set pointer to chb data
222         LDI @ENHIST,AR5         ;set pointer to energy history
223
224 ************************************************************************
225 **   INIT COMPLETE                                                     *
226 ************************************************************************
227
228
229 LOOP:   LDI     @READYA,R0              ; Wait for new samples to be available
230         LDI     @READYB,R1
231         ADDI    R0,R1                   ; check to see if both ready flags are set
232         LDI     2,R0
233         CMPI    R0,R1
234         BZ      BEGN_ALGR               ; go on to algorithm if both channels are ready
235         BR LOOP                         ; if data isn't ready then wait some more...
236
237 ************************************************************************
238 *  BEGINNING OF THE ALGORITHM                                           *
239 ************************************************************************
240
241 BEGN_ALGR       STI     0,@READYA       ;reset ready flags and go on to main algorithm
242                 STI     0,@READYB
243
244         LDI     @NEWSA,R0       ; load new channel a data
245         LDI     @NEWSB,R1       ; load new channel b data
246         NOP
247         STI     R0,@CSAMP       ;STORE CURRENT INT SAMPLE
248         FLOAT   R0,R0           ;convert to floating point
249         FLOAT   R1,R1           ;convert to floating point
250         NOP
251         LDF     1.0,R4
252         MPYF    R4,R0           ;
253         MPYF    R4,R1           ;
254
255         RND     R0,R0
256         RND     R1,R1
257
258        STF      R0,@OUTCHA      ;STORE CURRENT SAMPLE FOR POSS OUTPUT
259
260         MPYF    NNN,R0
261         MPYF    NNN,R0
262         MPYF    NNN,R1
263         MPYF    NNN,R1   ;bring sample values down in this area if needed
264
265         LDI     AR3,AR6
266         LDI     AR4,AR7
267         LDI     128,BK                  ;load size of circular buffers
268         LDI     64,IR1                  ;LOAD OFFSET FOR XCORR OLD VALUE
269         LDF     *AR6++(IR1)%,R2         ;get cha x(n-N) AND INCREMENT POINTER
270         LDF     *AR7++(IR1)%,R3         ;get chb x(n-N) AND INCREMENT POINTER
271         LDF     *AR6,R4                 ;get cha old value for xcorr
272         LDF     *AR7,R5                 ;get chb old value for xcorr
273
274         RND     R4,R4                   ;round all sample values
275         RND     R5,R5
276         RND     R2,R2
277         RND     R3,R3
```

```
278         RND     R0,R0
279         RND     R1,R1
280
281         STF     R4,@OLDCHAX     ;STORE OLD VALUE FOR XCORR
282         STF     R5,@OLDCHBX     ;STORE OLD VALUE FOR XCORR
283         STF     R2,@OLDCHA      ;store cha x(n-N) in variable
284         STF     R3,@OLDCHB      ;store chb x(n-N) in variable
285         STF     R0,@NEWCHA      ;STORE NEWEST CHA SAMPLE
286         STF     R1,@NEWCHB      ;STORE NEWEST CHB SAMPLE
287         STF     R0,*AR3++%      ;store current cha sample in circ. buff.
288 ||      STF     R1,*AR4++%      ;store current chb sample in circ. buff.
289



290 *********************************************
291 ***  start to compute chA frequencies       ***
292 *** calculate FIRST frequency for channel A ***
293 *********************************************
294
295         MPYF    ALPHAN,R2       ;R2=(alpha^128)*x(n-N)
296         SUBF3   R2,R0,R5        ;R5=x(n)-(alpha^128)*x(n-N)
297         LDF     @CHAF3R,R4      ;R4=CHAF3R
298         LDF     @CHAF3I,R6      ;R6=CHAF3I
299         MPYF    COS3,R4         ;R4=CHAF3R*COS(2*PI*126/128)
300         MPYF    SIN3,R6         ;R6=CHAF3I*SIN(2*PI*126/128)
301         ADDF    R6,R4           ;R4=CHAF3R*COS+CHAF3I*SIN
302         MPYF    ALPHA,R4        ;R4=ALPHA*(CHAF3R*COS+CHAF3I*SIN)
303         ADDF    R5,R4           ;R4= THE NEW CHAF3R
30
305 ** calculate the imaginary part ***
306
307         LDF     @CHAF3R,R2      ;R2=CHAF3R
308         LDF     @CHAF3I,R3      ;R3=CHAF3I
309         MPYF    COS3,R3         ;R3=CHAF3I*COS(2*PI*126/128)
310         MPYF    SIN3,R2         ;R2=CHAF3R*SIN(2*PI*126/128)
311         SUBF3   R2,R3,R2        ;R2=CHAF3R*SIN+CHAF3I*COS
312         MPYF    ALPHA,R2        ;R2=ALPHA*(CHAF3R*SIN+CHAF3I*COS)
313
314         RND     R4,R4
315         RND     R2,R2
316         STF     R4,@CHAF3R      ;STORE NEW CHAF3R
317         STF     R2,@CHAF3I      ;STORE NEW CHAF3I
318
319 *** calculate SECOND frequency for channel A ***
320
321         LDF     @CHAF2R,R4      ;R4=CHAF2R
322         LDF     @CHAF2I,R6      ;R6=CHAF2I
323         MPYF    COS2,R4         ;R4=CHAF2R*COS(2*PI*127/128)
324         MPYF    SIN2,R6         ;R6=CHAF2I*SIN(2*PI*127/128)
325         ADDF    R6,R4           ;R4=CHAF2R*COS+CHAF2I*SIN
326         MPYF    ALPHA,R4        ;R4=ALPHA*(CHAF2R*COS+CHAF2I*SIN)
327         ADDF    R5,R4           ;R4= THE NEW CHAF2R
328
329 ** calculate the imaginary part ***
330
331         LDF     @CHAF2R,R2      ;R2=CHAF2R
332         LDF     @CHAF2I,R3      ;R3=CHAF2I
333         MPYF    COS2,R3         ;R3=CHAF2I*COS(2*PI*127/128)
334         MPYF    SIN2,R2         ;R2=CHAF2R*SIN(2*PI*127/128)
335         SUBF3   R2,R3,R2                ;R2=CHAF2R*SIN+CHAF2I*COS
336         MPYF    ALPHA,R2        ;R2=ALPHA*(CHAF2R*SIN+CHAF2I*COS)
337
338         RND     R4,R4
339         RND     R2,R2
340         STF     R4,@CHAF2R      ;STORE NEW CH1F2R
341         STF     R2,@CHAF2I      ;STORE NEW CH1F2I
342
343
344 *********************************************
345 ***  start to compute chB frequencies       ***
346 *** calculate FIRST frequency for channel B ***
```

```
347 ********************************************
348
349         LDF     @OLDCHB,R2      ;GET x(n-N) for chB
350         MPYF    ALPHAN,R2       ;R2=(alpha^256)*x(n-N)
351         SUBF3   R2,R1,R5        ;R5=x(n)-(alpha^256)*x(n-N)
352         LDF     @CHBF3R,R4      ;R4=CHBF3R
353         LDF     @CHBF3I,R6      ;R6=CHBF3I
354         MPYF    COS3,R4         ;R4=CHBF3R*COS(2*PI*126/128)
355         MPYF    SIN3,R6         ;R6=CHBF3I*SIN(2*PI*126/128)
356         ADDF    R6,R4           ;R4=CHBF3R*COS+CHBF3I*SIN
357         MPYF    ALPHA,R4        ;R4=ALPHA*(CHBF3R*COS+CHBF3I*SIN)
358         ADDF    R5,R4           ;R4= THE NEW CHBF3R
359
360 ** calculate the imaginary part ***
361
362         LDF     @CHBF3R,R2      ;R2=CHBF3R
363         LDF     @CHBF3I,R3      ;R3=CHBF3I
364         MPYF    COS3,R3         ;R3=CHBF3I*COS(2*PI*126/128)
365         MPYF    SIN3,R2         ;R2=CHBF3R*SIN(2*PI*126/128)
366         SUBF3   R2,R3,R2        ;R2=CHBF3R*SIN+CHBF3I*COS
367         MPYF    ALPHA,R2        ;R2=ALPHA*(CHBF3R*SIN+CHBF3I*COS)
368
369         RND     R4,R4
370         RND     R2,R2
371         STF     R4,@CHBF3R      ;STORE NEW CHBF3R
372         STF     R2,@CHBF3I      ;STORE NEW CHBF3I
373
374 *** calculate SECOND frequency for channel B ***
375
376         LDF     @CHBF2R,R4      ;R4=CHBF2R
377         LDF     @CHBF2I,R6      ;R6=CHBF2I
378         MPYF    COS2,R4         ;R4=CHBF2R*COS(2*PI*127/128)
379         MPYF    SIN2,R6         ;R6=CHBF2I*SIN(2*PI*127/128)
380         ADDF    R6,R4           ;R4=CHBF2R*COS+CHBF2I*SIN
381         MPYF    ALPHA,R4        ;R4=ALPHA*(CHBF2R*COS+CHBF2I*SIN)
382         ADDF    R5,R4           ;R4= THE NEW CHBF2R
383
384 ** calculate the imaginary part ***
385
386         LDF     @CHBF2R,R2      ;R2=CHBF2R
387         LDF     @CHBF2I,R3      ;R3=CHBF2I
388         MPYF    COS2,R3         ;R3=CHBF2I*COS(2*PI*127/128)
389         MPYF    SIN2,R2         ;R2=CHBF2R*SIN(2*PI*127/128)
390         SUBF3   R2,R3,R2        ;R2=CH1F2R*SIN+CHBF2I*COS
391         MPYF    ALPHA,R2        ;R2=ALPHA*(CHBF2R*SIN+CHBF2I*COS)
392
393         RND     R4,R4
394         RND     R2,R2
395         STF     R4,@CHBF2R      ;STORE NEW CHBF2R
396         STF     R2,@CHBF2I      ;STORE NEW CHBF2I
397
398
399
400 ************************************************************
401 *** calculate the square of the magnitudes of the frequencies ***
402 ************************************************************
403
404         MPYF    R4,R4           ;CHBF2R^2
405         MPYF    R2,R2           ;CHBF2I^2
406         ADDF    R4,R2           ;CHBF2R^2+CHBF2I^2
407         LDF     @CHBF3R,R4      ;GET CHBF3R
408         ABSF    R2,R2
409         STF     R2,@MCHBF2      ;STORE MCHBF2
410         MPYF    R4,R4           ;CHBF3R^2
411         LDF     @CHBF3I,R2      ;GET CHBF3I
412         MPYF    R2,R2           ;CHBF3I^2
413         ADDF    R4,R2           ;CHBF3R^2+CHBF3I^2
414         LDF     @CHAF3R,R4      ;GET CHAF3R
415         ABSF    R2,R2           ;ABS. VALUE OF MCHBF3
416         STF     R2,@MCHBF3      ;STORE MCHBF3
417         MPYF    R4,R4           ;CHAF3R^2
418         LDF     @CHAF3I,R2      ;GET CHAF3I
419         MPYF    R2,R2           ;CHAF3I^2
```

```
420         ADDF    R4,R2           ;CHAF3R^2+CHAF3I^2
421         LDF     @CHAF2R,R4      ;GET CHAF2R
422         ABSF    R2,R2           ;ABS. VALUE OF MCHAF3
423         STF     R2,@MCHAF3      ;STORE MCHAF3
424         MPYF    R4,R4           ;CHAF2R^2
425         LDF     @CHAF2I,R2      ;GET CHAF2I
426         MPYF    R2,R2           ;CHAF2I^2
427         ADDF    R4,R2           ;CHAF2R^2+CHAF2I^2
428         ABSF    R2,R2           ;ABS. VALUE OF MCHAF2
429         STF     R2,@MCHAF2      ;STORE MCHAF2
430
431
432 ****************************************
433 ***       calculate energy function     ***
434 ****************************************
435
436         LDF     @MCHBF2,R3      ;GET MCHBF2
437         LDF     @MCHAF3,R4      ;GET MCHAF3
438         MPYF    R3,R2           ;R2=MCHAF2*MCHBF2
439         LDF     @MCHBF3,R3      ;GET MCHBF3
440         MPYF    R4,R3           ;R3=MCHAF3*MCHBF3
441         ADDF    R3,R2           ;R2=MCHAF3*MCHBF3+MCHAF2*MCHBF2
442         RND     R2,R2           ;ROUND BEFORE STORING
443         STF     R2,@CUREF       ;STORE CURRENT ENERGY FUNCTION
444
445 ****************************************
446 *** FIND MAX FROM OLDEST EF TO NEWEST-70  ***
447 ****************************************
448
449         LDI     690,BK         ;load block size
450         LDI     AR5,AR6         ;TRANSFER TO A DUMMY POINTER
451         LDF     0.0,R3          ;INITIALIZE VARIABLE
452         LDI     619,RC          ;REPEAT BLOCK 620 TIMES
453         RPTB    ENDLOOP
454         LDF     *AR6++%,R2      ;GET EF VALUE
455         CMPF    R3,R2           ;IF JUST LOADED EF < MAX FOUND SO FAR THEN GOTO END
456         BN      ENDLOOP
457         LDF     R2,R3           ;A NEW MAX FOUND, STORE IT
458 ENDLOOP NOP
459         LDF     R3,R0           ;LOAD MAX INTO R0 TO FACILITATE INVERSE CALC.
460         CALL    fpinv           ;CALCULATE 1/MAX
461         LDF     @CUREF,R2       ;GET CURRENT EF VALUE
462         MPYF    R2,R0           ;CUREF/MAX
463         STF     R0,@DETFUN      ;STORE VALUE
464         STF     R2,*AR5++%      ;STORE CURRENT ENERGY FUCNTION IN HISTORY
465
466
467 ************************************
468 *** calculate zero lag correlation ***
469 ************************************
470
471 *** calculate autocorrelation of chA ***
472
473         LDF     @OLDCHAX,R2     ;GET X(n-N) OF CHA
474         RND     R2,R2
475         MPYF    R2,R2           ;R2=CHA(n-N)^2
476         ABSF    R2,R2           ;TAKE ABS.
477         LDF     @NEWCHA,R3      ;GET X(n) OF CHA
478         RND     R3,R3
479         MPYF    R3,R3           ;R3=CHA(n)^2
480         ABSF    R3,R3           ;TAKE ABS.
481         LDF     @ACA,R4         ;GET PREVIOUS AUTOCORR OF CHA
482         RND     R4,R4
483         ADDF    R3,R4           ;ADD NEW UPDATE TO PREVIOUS
484         SUBF    R2,R4           ;SUB OLD OUT
485         ABSF    R4,R4           ;TAKE ABS VALUE
486         RND     R4,R4
487         STF     R4,@ACA         ;STORE NEW AUTOCORR OF CHA
488
489 *** calculate autocorrelation of chB ***
490
491         LDF     @OLDCHBX,R2     ;GET X(n-N) OF CHB
492         RND     R2,R2
```

```
493         MPYF    R2,R2           ;R2=CHA(n-N)^2
494         ABSF    R2,R2           ;TAKE ABS.
495         LDF     @NEWCHB,R3      ;GET X(n) OF CHB
496         RND     R3,R3
497         MPYF    R3,R3           ;R3=CHA(n)^2
498         ABSF    R3,R3           ;TAKE ABS.
499         LDF     @ACB,R4         ;GET PREVIOUS AUTOCORR OF CHB
500         RND     R4,R4
501         ADDF    R3,R4           ;ADD NEW UPDATE TO PREVIOUS
502         SUBF    R2,R4           ;SUB OLD OUT
503         ABSF    R4,R4           ;TAKE ABS VALUE
504         RND     R4,R4
505         STF     R4,@ACB         ;STORE NEW AUTOCORR OF CHB
506
507
508 *** calculate cross-correlation of chA and chB ***
509
510         LDF     @OLDCHAX,R2     ;GET X(n-N) OF CHA
511         RND     R2,R2
512         LDF     @OLDCHBX,R3     ;GET X(n-N) OF CHB
513         RND     R3,R3
514         MPYF3   R2,R3,R4        ;R4=CHA(n-N)*CHB(n-N)
515         LDF     @NEWCHA,R2      ;GET X(n) OF CHA
516         RND     R2,R2
517         LDF     @NEWCHB,R3      ;GET X(n) OF CHB
518         RND     R3,R3
519         MPYF    R2,R3           ;R3=CHA(n)*CHB(n)
520         LDF     @XC,R2          ;GET PREVIOUS X-CORR
521         RND     R2,R2
522         ADDF    R3,R2           ;ADD IN  UPDATE
523         SUBF    R4,R2           ;SUB OUT OLD
524         RND     R2,R2
525         STF     R2,@XC          ;STORE NEW X-CORR
526
527         ABSF    R2,R2
528         MPYF3   R2,R2,R6        ;R6=XCORR^2
529         LDF     @ACA,R2         ;GET AUTOCORR CHA
530         RND     R2,R2
531         LDF     @ACB,R3         ;GET AUTOCORR CHB
532         RND     R3,R3
533         MPYF3   R2,R3,R0        ;R0=ACA*ACB
534         CALL    fpinv           ;COMPUTE 1/R0
535         MPYF    R6,R0           ;R0=XCORR^2/(AUTOCORRA*AUTOCORRB)
536         RND     R0,R0
537         STF     R0,@ZC          ;STORE NEW ZERO-LAG X-CORR
538
539 ************************************************************************
540 *  END OF THE ALGORITHM                                                *
541 ************************************************************************
542
543 ************************
544 *** DECISION SECTION ****
545 ************************
546
547 *** COMPARE CURRENT DETERMINATION FUNCTION TO THRESHOLD ***
548
549 DFCHK   LDF     @DETFUN,R1      ;GET CURRENT DETFUN (decision ratio) VALUE
550         LDF     25,R2           ;R2=25.0
551         CMPF    R1,R2           ;COMPARE detfun to 25
552         BN      STAGE1          ;GO TO STAGE1 IF DETFUN>25
553         BZ      STAGE1          ;GO TO STAGE1 IF DETFUN=25
554         BR      NOOUT           ;Go to no output stage if <25
555
556 *** STAGE1: AT THIS STAGE DETFUN>=25, NOW CHECK CORRELATION ***
557
558 STAGE1  NOP
559         LDF     @ZC,R0          ;GET ZEROTH LAG CORRELATION VALUE
560         LDF     ZCTHR,R1        ;GET CORRELATION THRESHOLD
561         CMPF    R0,R1           ;COMPARE
562         BN      OUTSTAGE        ;IF ZC>ZCTHR THEN GOTO OUTSTAGE
563         BR      NOOUT           ;GO TO NO output stage if ZC<ZCTHR
564
565 *** OUTSTAGE: THERE IS A BLAST, NOW OUTPUT THE CURRENT SAMPLE ***
```

```
566
567 OUTSTAGE        NOP
568         LDI     1,R2            ;output a 1 because there's a blast
569         STI     R2,*AR2         ;write 1 to comm port 3 output to processor A
570         BR      LOOP            ;go back and wait for next set of data points
571
572 *** NOOUT: THERE IS NO BLAST AND DECAY WAS NOT ACTIVE, OUTPUT 0 ***
573
574 NOOUT   LDI 0,R0
575         STI R0,*AR2             ;OUTPUT 0 TO processor 1 over comm port 3
576         BR  LOOP                ;go back and wait for next set of data points
577
578
579 ************************************************
580 *** subroutine Inverse of a floating point number ***
581 *** on page 11-29 in TI C30 User's Guide          ***
582 *** INPUT AND OUTPUT IS IN R0                     ***
583 *** USES R0,R1,R2,R3                              ***
584 ************************************************
585
586 fpinv:  LDF     R0,R3
587         ABSF    R0
588
589         PUSHF   R0
590         POP     R1
591         ASH     -24,R1
592
593         NEGI    R1
594         SUBI    1,R1
595         ASH     24,R1
596         PUSH    R1
597         POPF    R1
598
599         MPYF    R1,R0,R2
600         SUBRF   2.0,R2
601         MPYF    R2,R1
602
603         MPYF    R1,R0,R2
604         SUBRF   2.0,R2
605         MPYF    R2,R1
606
607         MPYF    R1,R0,R2
608         SUBRF   2.0,R2
609         MPYF    R2,R1
610
611         MPYF    R1,R0,R2
612         SUBRF   2.0,R2
613         MPYF    R2,R1
614
615        RND     R1
616
617         MPYF    R1,R0,R2
618         SUBRF   1.0,R2
619         MPYF    R1,R2
620         ADDF    R2,R1
621
622        RND     R1,R0
623
624         NEGF    R0,R2
625         LDF     R3,R3
626         LDFN    R2,R0
627
628         RETS
629 ***********************************
630 *** END OF SUBROUTINE FPINV      ***
631 ***********************************
632
633 ***************************************************************************
634 *  Interrupt service routine for reading port3 (get cha data)              *
635 ***************************************************************************
636
637 ICRDY3: LDI     *AR0,R10        ; read new cha sample
638         STI     R10,@NEWSA      ; store it
```

```
639         STI     1,@READYA       ; set ready flag
640         RETI
641
642 ************************************************************************
643 *  Interrupt service routine for reading port4 (get chb data)           *
644 ************************************************************************
645
646 ICRDY4: LDI     *AR1,R11        ; read new chb sample
647         STI     R11,@NEWSB      ; store it
648         STI     1,@READYB       ; set ready flag
649         RETI
650
651 ****************************************************************
652 *      END OF PROGRMAM                                         *
653 ****************************************************************
654
655
656         .end
```

Appendix D2: MON7_1 code (CPU_A)
Noise Monitor Computations

```
 1 ********************************************************************
 2 *                                                                  *
 3 *                                                                  *
 4 *   Date     : 11 SEP 1996                                         *
 5 *   Title    : MON7_1.ASM                                          *
 6 *                                                                  *
 7 *   MON7_1 initializes the AM/D16DS daughter module to sample      *
 8 *   at 48KHz, and passes the samples to the second C40 processor.  *
 9 *   Decimation for processor B is done here.                       *
10 *                                                                  *
11 *       For C-filter:                                              *
12 * Biquad0=0.44519794+0.89039588*(Z^-1)+0.44519794*(Z^-2)          *
13 *         ____________________________________________             *
14 *         1-0.2231468*(Z^-1)+0.00012448623*(Z^-2)                  *
15 *                                                                  *
16 * Biquad1=0.44519794-0.89039588*(Z^-1)+0.44519794*(z^-2)          *
17 *         ____________________________________                     *
18 *         1-1.9946068*(Z^-1)+0.99461403*(Z^-2)                     *
19 *                                                                  *
20 *       with gain factor correction of 1.3101327                   *
21 *   MON7_1 is part of the two processor noise monitor prototype    *
22 *   and should be loaded on the primary processor module.  MON7_2  *
23 *   has to be loaded on the secondary processor module, and        *
24 *   contains the blast recognition code. This revision differs     *
25 *   from mon6_1 in that this code uses cross multiplication        *
26 *   in the measurement of the signal to improve the accuracy in    *
27 *   the presence of wind.                                          *
28 *                                                                  *
29 ********************************************************************
30
31
32                 .data
33 C_C             .float  -0.00012448623  ; a2 for biquad 0
34                 .float  0.44519794      ; b2 for    “
35                 .float  0.2231468       ; a1 for    “
36                 .float  0.89039588      ; b1 for    “
37                 .float  0.44519794      ; b0 for    “
38
39                 .float  -0.99461403     ; a2 for biquad 1
40                 .float  0.44519794      ; b2 for    “
41                 .float  1.9946068       ; a1 for    “
42                 .float  -0.89039588     ; b1 for    “
43                 .float  0.44519794      ; b0 for    “
44
45                 .space 2                ; spacer in order to start
46                                         ; delays on address which is Mult of 4
47
48
49                 .float 0.0              ; 3 locations for biquad 0 delays cha
50                 .float 0.0
51 C_DA            .float 0.0
52
53                 .space 1                ; once again a spacer to keep data in right place
54
55                 .float 0.0              ; 3 locations for biquad 1 delays cha
56         .float 0.0
57         .float 0.0
58
```

```
59         .space 1
60
61                  .float 0.0              ; 3 locations for biquad 0 delays chb
62                  .float 0.0              ;
63 C_DB             .float 0.0
64
65                 .space 1
66
67                  .float  0.0             ; 3 locations for biquad 1 delays chb
68         .float  0.0
69         .float  0.0
70
71         ; note that a2 and a1 here are the
72         ; negative of the a2 and a1 given
73         ; by MATLAB in its filter design
74         ; due to a difference in a definition
75
76
77 STACK            .word   002ffc00H       ; Define stack space
78 DATA_P           .word   00300000H       ; define data page pointer location
79 IACKLOC          .word   80000000H       ; Interrupt acknowledge location
80 IVECTAB          .word   00308000H       ; Interrupt vector table
81 CHANA            .word   90000002H       ; Channel A address
82 CHANB            .word   90000006H       ; Channel B address
83 UCR              .word   90000008H       ; User control register address
84 ACR              .word   9000000aH       ; Analog control register address
85 IMR              .word   9000000bH       ; Analog interrupt mask register addr
86 CONFIG           .word   9000000fH       ; Configuration register address
87 COMM_PORT0_CTL   .word   00100040h       ; Port 0 control
88 COMM_PORT0_OTP   .word   00100042h       ;        output
89 COMM_PORT0_INP   .word   00100041h       ;        input
90 COMM_PORT1_CTL   .word   00100050h       ; Configuration register address
91 COMM_PORT1_INP   .word   00100051h       ;        input
92 COMM_PORT1_OTP   .word   00100052h       ;        output
93 DPRAM_ADDR       .word   0a0000000h      ; start address of dual port ram
94 CURRENT_ADDR     .word   0a0000400h      ; address of spot where current circ buff address
95                                          ; for current TSB is stored
96 FIRST_TIME       .word   1h              ; set first time flag to one
97 DEC_COUNT        .word   0h              ; decimation counter
98 CHA_C            .float  0               ; current C-filtered data cha
99 CHB_C            .float  0               ; current C-filtered data chb
100 C_COEFF         .word   C_C             ; address of C filter coeff.
101 C_DELAYA        .word   C_DA            ; address of C filter delays for ch a
102 C_DELAYB        .word   C_DB            ; address of C filter delays for ch b
103 GAIN_CON        .float  1.3101327       ; c-filter gain correction constant
104
105 MICRO_BLK_CNT    .word   0       ; counter for micro blocks (0-199)
106
107 SQA_MICRO_BLK   .float  0       ; sum of squares for current micro block, cha
108 SQAC_MICRO_BLK  .float  0       ; "    "    "    "     "      "     "  , cha c-wght
109 SQB_MICRO_BLK   .float  0       ; "    "    "    "     "      "     "  , chb
110 SQBC_MICRO_BLK  .float  0       ; "    "    "    "     "      "     "  , chb c-wght
111 SQAB_MICRO_BLK  .float  0       ; "    "    "    "     "      "     "  , cha*chb
112 SQABC_MICRO_BLK .float  0       ; "    "    "    "     "      "     "  , cha*chb c-wght
113
114 MICRO_MAX_A     .float  0       ; current max val for current micro block, cha
115 MICRO_MAX_AC    .float  0       ;    "     "   "   "     "      "      " , cha c-wght
116 MICRO_MAX_B     .float  0       ;    "     "   "   "     "      "      " , chb
117 MICRO_MAX_BC    .float  0       ;    "     "   "   "     "      "      " , chb c-wght
118 MICRO_MAX_AB    .float  0       ;    "     "squared val for current m_blk, cha*chb
119 MICRO_MAX_ABC   .float  0       ;    "     "   "     "   "      "      "  , cha*chb c-wght
120
121 SEND_DATA_ADDR  .word   00300400h               ; address that corresponds to SQA_TSB
122 CHA_INTEGER     .word   0
123 CHB_INTEGER     .word   0
124 BLR_FLAG        .word   0       ; current blr flag
125
126         .sect "to_send"
127
128 ************************************************************
129 *  This is data that is sent to the PC via DPRAM                *
130 *  the order here is important as indirect addressing is used to *
131 *  get the data before writing to the DPRAM                     *
```

```
132 ****************************************************************
133
134 SQA_TSB          .float  0       ; sum of squares for current tenth sec blk, cha
135 SQAC_TSB         .float  0       ;  “   “    “     “     “      “    “   “ , cha c-wght
136 SQB_TSB          .float  0       ;  “   “    “     “     “      “    “   “ , chb
137 SQBC_TSB         .float  0       ;  “   “    “     “     “      “    “   “ , chb c-wght
138
139 TSB_MAX_A        .float  0       ; current max val for current tenth sec blk, cha
140 TSB_MAX_AC       .float  0       ;    “     “   “   “     “      “    “   “ , cha c-wght
141 TSB_MAX_B        .float  0       ;    “     “   “   “     “      “    “   “ , chb
142 TSB_MAX_BC       .float  0       ;    “     “   “   “     “      “    “   “ , chb c-wght
143
144 TSB_MAX_POS_A    .word   0       ; position of max in TSB to 1/200 of a TSB , cha
145 TSB_MAX_POS_AC   .word   0       ;     “     “  “   “   “    “    “ “  “  , cha c-wght
146 TSB_MAX_POS_B    .word   0       ;     “     “  “   “   “    “    “ “  “  , chb
147 TSB_MAX_POS_BC   .word   0       ;     “     “  “   “   “    “    “ “  “  , chb c-wght
148
149 TSB_NUM_BLR      .word   0       ; number of blast recognitions in current TSB
150
151 SQAB_TSB         .float  0       ; sum of squares for current tenth sec blk, cha*chb
152 SQABC_TSB        .float  0       ;  “   “    “     “     “      “    “   “ , cha*chb c-wght
153 TSB_MAX_AB       .float  0       ; current max square val for tenth sec blk, cha*chb
154 TSB_MAX_ABC      .float  0       ; current max square val for tenth sec blk, cha*chb c-wght
155 TSB_MAX_POS_AB   .word   0       ; position of max in TSB to 1/200 of a TSB, cha*chb
156 TSB_MAX_POS_ABC  .word   0       ; position of max in TSB to 1/200 of a TSB, cha*chb c-wght
157
158
159                .sect “DP_circ”
160
161                .space 950              ;zero out circular buffer in DPRAM length=3B6h
162
163                .text
164 ************************************
165 *  start of program text           *
166 ************************************
167                                        ; Set up interrupt vector table
168                 BR      START
169                 .word   0H             ; NMI   (Unused interrupts)
170                 .word   0H             ; TINT0
171                 .word   0H             ; IIOF0
172                 .word   ANALOG         ; IIOF1 - Amelia interrupt
173                 .word   0H             ; IIOF2
174                 .word   0H             ; IIOF3
175
176                 .space  6              ; Reserved space
177
178                 .word   0h             ; Input channel full, port 0
179                 .word   0h             ; Input channel ready, ICRDY0
180                 .word   0h             ; Output channel ready
181                 .word   0h             ; Output channel empty
182                 .word   0h             ; input channel full, port 0
183                 .word   0h             ; input channel ready
184                 .word   0h             ; output channel ready
185                 .word   0h             ; output channel empty
186
187         .global START                  ; set up global variables
188         .global WAIT
189         .global LOOPA
190         .global LOOPB
191         .global ANALOG
192         .global ICRDY0
193         .global CHA_INTEGER
194         .global CHB_INTEGER
195         .global SEND_DATA
196         .global DEC_COUNT
197         .global NO_TRNFR
198         .global FILTER
199         .global C_COEFF
200         .global C_DELAYA
201         .global C_DELAYB
202         .global C_C
203         .global C_DA
204         .global C_DB
```

```
205         .global GO_ON1
206         .global GO_ON2
207         .global GO_ON3
208         .global GO_ON4
209         .global GO_ON5
210         .global GO_ON6
211         .global GO_ON7
212         .global GO_ON8
213         .global GO_ON9
214         .global GO_ON10
215         .global GO_ON11
216         .global GO_ON12
217         .global GO_ON13
218         .global GO_ON14
219         .global GO_ON15
220         .global GO_ON16
221         .global GO_ON17
222         .global GO_ON18
223         .global GET_BLR
224         .global END_OF_ISR
225         .global END_OF_TSB
226         .global SQA_MICRO_BLK
227         .global SQAC_MICRO_BLK
228         .global SQB_MICRO_BLK
229         .global SQBC_MICRO_BLK
230         .global SQAB_MICRO_BLK
231         .global SQABC_MICRO_BLK
232         .global MICRO_MAX_A
233         .global MICRO_MAX_AC
234         .global MICRO_MAX_B
235         .global MICRO_MAX_BC
236         .global MICRO_MAX_AB
237         .global MICRO_MAX_ABC
238         .global SQA_TSB
239         .global SQAC_TSB
240         .global SQB_TSB
241         .global SQBC_TSB
242         .global SQAB_TSB
243         .global SQABC_TSB
244         .global TSB_MAX_A
245         .global TSB_MAX_AC
246         .global TSB_MAX_B
247         .global TSB_MAX_BC
248         .global TSB_MAX_POS_A
249         .global TSB_MAX_POS_AC
250         .global TSB_MAX_POS_B
251         .global TSB_MAX_POS_BC
252         .global TSB_MAX_POS_AB
253         .global TSB_MAX_POS_ABC
254         .global TSB_NUM_BLR
255         .global SEND_DATA_ADDR
256         .global DPRAM_ADDR
257         .global CURRENT_ADDR
258         .global COMM_PORT0_CTL
259         .global COMM_PORT1_CTL
260         .global COMM_PORT0_INP
261         .global COMM_PORT0_OTP
262         .global COMM_PORT1_OTP
263         .global MICRO_BLK_CNT
264         .global READ
265         .global CONT1
266         .global CONT2
267         .global CONT3
268         .global FIRST_TIME
269         .global BLR_FLAG
270
271
272
273                                         ; Start of program...
274 START:          LDP     @DATA_P,DP      ; Initialize the stack
275                LDI     @STACK,SP
276
277                LDI     @IVECTAB,R0     ; Set up the interrupt vector table
```

```
278                 LDPE    R0,IVTP
279
280 ;Write to the registers within AMELIA...
281
282                 LDI     @UCR,AR0        ; User control register
283                 LDI     0a000H,R5       ;
284                 STI     R5,*AR0         ; ADMCLK1 to be used
285
286                 LDI     @ACR,AR0        ; Analog control register
287                 LDI     0a0H,R5         ;
288                 STI     R5,*AR0         ; 48 kHz sample rate
289                                         ; AMELIA into reset
290
291                 LDI     @ACR,AR0        ; Analog control register
292                 LDI     0e0H,R5         ;
293                 STI     R5,*AR0         ; AMELIA released from reset,
294                                         ; calibrating
295
296                 LDI     @CONFIG,AR0     ; Analog configuration register
297                 LDI     0b390H,R5       ; loaded with Key value
298                 STI     R5,*AR0         ;
299
300                 LDI     @IMR,AR0        ; Analog interrupt mask register
301                 LDI     01H,R5          ; Int when RX register full
302                 STI     R5,*AR0         ;
303
304                 LDI     @IMR,AR0         ; AR0 = Interrupt mask register
305                 LDI     @CHANA,AR1       ; AR1 = channel a input
306                 LDI     @IACKLOC,AR2     ; AR2 = Interrupt acknowledge location
307                 LDI     @CHANB,AR3       ; AR3 = channel b input
308
309 ; Amelia running and ready.  Initialize comm port pointers...
310
311                 LDA     @COMM_PORT0_OTP, AR4    ;       output FIFO addr -channel a
312                 LDA     @COMM_PORT0_INP, AR5    ;       input FIFO addr
313                 LDA     @COMM_PORT1_OTP, AR6    ;       output FIFO addr -channel b
314                 LDA     @DPRAM_ADDR, AR7        ;       dual port ram
315
316 ; zero out the input comm port0 buffer
317
318                 PUSH    AR4                     ; save ar4 on stack
319                 LDA     @COMM_PORT0_CTL,AR4     ; load address of port0 control reg.
320                 LBU1    *AR4,R1                 ; load byte 1 of control reg
321                 LSH     3,R1                    ; shift left by 3 bits
322                 LBU0    R1,R1                   ; take only byte 0 of R1
323                 LSH     -4,R1                   ; Right shift by 4
324     ; now R1 contains number of words
325     ; in the port0 input buffer
326                 POP     AR4                     ; restore ar4
327                 LDI     0,R2
328                 CMPI    R1,R2                   ; compare #words to zero
329                 BZ      CONT1                   ; if no data in register then continue on
330                 ADDI    -1,R1                   ; subtract one from #words
331                 LDI     R1,RC                   ; load repeat counter with words-1
332                 RPTB    READ
333 READ            LDI     *AR5,R3                 ; read in data until its all gone
334
335 ; Enable interrupts...
336
337 CONT1           OR      0B0h, IIF               ; IIOF1 interrupt -(A/D board)
338                 OR      02000h, ST              ; CPU global interrupt
339                 IACK    *AR2
340
341 WAIT:           BR      WAIT                    ; IDLE until interrupted
342
343
344 ************************************************************************
345 *  Interrupt service routine for reading the analog data from the ADC  *
346 *  and passing to processor B, filtering and monitor calculations.     *
347 *  Sends data at a rate of 2kHz and only if the output FIFO buffer is empty *
348 ************************************************************************
349
350 ANALOG:
```

```
351                 LDI     *AR0,R11                ; Read Amelia interrupt to clear
352
353                 LDI     *AR1,R10                ; Load channel a input -> R10
354                 LDI     *AR3,R11                ; load channel b input -> R11
355                 STI     R10,@CHA_INTEGER        ; store cha integer sample value
356                 STI     R11,@CHB_INTEGER        ; store chb integer sample value
357                 LSH     16,R10                  ; left shift by 16 bits
358                 LSH     16,R11
359
360                 LDI     @DEC_COUNT,R8           ; load decimation counter
361                 LDI     23,R9                   ; we are taking every 24th sample for proc. B
362                 CMPI    R8,R9                   ; compare dec_count to 23
363                 BZ      SEND_DATA               ; goto "send_data" if counter=23
364                 ADDI    1,R8                    ; increment dec_count by one
365                 STI     R8,@DEC_COUNT
366                 BR      FILTER
367
368 SEND_DATA       PUSH    AR1                     ; push address register onto stack
369                 PUSH    AR3                     ; push address register onto stack
370                 LDA     @COMM_PORT0_CTL,AR1     ; load address of port0 control reg.
371                 LDA     @COMM_PORT1_CTL,AR3     ; load address of port1 control reg.
372                 LBU0    *AR1,R8                 ; load byte 0 of comm_port0 control reg.
373                 LSH     -4,R8                   ; right shift by 4 bits
374                                                 ; R8=# of words in output FIFO
375                 LDI     0,R9
376                 CMPI    R8,R9                   ; compare # words in FIFO to zero
377                 BN      NO_TRNFR                ; if #words >0 then no transfer of data
378                                                 ; (allow proc. B to get in sync.
379                 STI     R10,*AR4                ; send cha data to proc. B over comm0
380                 STI     R11,*AR6                ; send chb data to proc. B over comm1
381                 STI     R9,@DEC_COUNT           ; zero the decimation counter
382                 POP     AR3                     ; restore AR3
383                 POP     AR1                     ; restore AR1
384                 BR      FILTER
385
386
387 NO_TRNFR        STI     R9,@DEC_COUNT           ; reset the decimation counter to 0
388                 POP     AR3                     ; restore AR3
389                 POP     AR1                     ; restore AR1
390                 BR      FILTER
391
392
393 FILTER          LDI     R10,R2                  ; copy cha input to R2
394                 FLOAT   R2,R2                   ; convert to floating point
395
396                 PUSH    AR0                     ; save important registers on stack
397                 PUSH    AR1
398                 PUSH    BK
399                 PUSH    IR0
400                 PUSH    IR1
401                 PUSH    RC
402
403 ********************************************************************************
404 *  FILTER CH A with IIR C-filter                                               *
405 * (Basic IIR filter routine from TI TMS320C4x User's Guide)                    *
406 ********************************************************************************
407
408                 LDA     @C_COEFF,AR0            ; load address of filter coefficients ( a2 )
409                 LDA     @C_DELAYA,AR1           ; load address of filter delay nodes ( d(n-2) )
410                 LDI     3,BK                    ; initialize Block-size to 3
411                 LDI     4,IR0
412                 LDI     4,IR1
413                 LDI     0,RC
414
415                 MPYF3   *AR0,*AR1,R0            ; a2(0)*d(0,n-2) -> R0
416                 MPYF3   *AR0++(1),*AR1--(1)%,R1 ; b2(0)*d(0,n-2) -> R1
417
418                 RPTBD   LOOPA
419
420                 MPYF3   *++AR0(1),*AR1,R0       ;a1(0)*d(0,n-1) -> R0
421 ||              ADDF    R0,R2,R2                ;First sum term of d(0,n)
422
423                 MPYF3   *++AR0(1),*AR1--(1)%,R0 ;b1(0)*d(0,n-1) -> R0
```

```
424 ||              ADDF3   R0,R2,R2                ;Second sum term of d(0,n)
425                 MPYF3   *++AR0(1),R2,R2         ;b0(0)*d(0,n) -> R2
426 ||              STF     R2,*AR1--(1)%           ;store d(0,n) point to d(0,n-2)
427
428
429                                                 ; loop starts here
430                 MPYF3   *++AR0(1),*++AR1(IR0),R0        ; a2(i)*d(i,n-2) ->R0
431 ||              ADDF3   R0,R2,R2                ; First sum term of y(i-1,n)
432                                                 ; Pipeline hit on previous
433                                                 ; instruction
434
435                 MPYF3   *++AR0(1),*AR1--(1)%,R1         ; b2(i)*D(i,n-2) -> R1
436 ||              ADDF3   R1,R2,R2                ; Second sum term of y(i-1,n)
437                 MPYF3   *++AR0(1),*AR1,R0               ; a1(i)*d(i,n-1) -> R0
438 ||              ADDF3   R0,R2,R2                ; First sum term of d(i,n)
439
440                 MPYF3   *++AR0(1),*AR1--(1)%,R0         ; b1(i)*d(i,n-1) ->R0
441 ||              ADDF3   R0,R2,R2                ; Second sum term of d(i,n)
442
443 LOOPA           MPYF3   *++AR0(1),R2,R2                 ; b0(i)*d(i,n) ->R2
444 ||              STF     R2,*AR1--(1)%                   ; store d(i,n) point to d(i,n-2)
445
446                                                 ; final summation
447
448                 ADDF    R0,R2                  ; first sum term of y(n-1,n)
449                 ADDF3   R1,R2,R0                ; second sum term of y(n-1,n)
450                 LDI     *AR1--(IR1),R1          ; return to first biquad
451                 LDI     *AR1--(1)%,R2           ; point to d(0,n-1)
452
453
454                 STI     AR1,@C_DELAYA           ; store updated delay pointer
455
456 ************************************************************************
457 *    end of filtering for CH A                                         *
458 ************************************************************************
459
460                 LDF     @GAIN_CON,R2            ; load the filter gain correction const.
461                 MPYF    R2,R0                   ; multiply result by gain const
462                 STF     R0,@CHA_C               ; store current c-filtered ch A sample
463
464                 LDI     R11,R2                  ; copy chb input to R2
465                 FLOAT   R2,R2                   ; convert to floating point
466
467 ************************************************************************
468 *   FILTER CH B with IIR C-filter                                      *
469 ************************************************************************
470
471                 LDA     @C_COEFF,AR0            ; load address of filter coefficients ( a2 )
472                 LDA     @C_DELAYB,AR1           ; load address of filter delay nodes ( d(n-2) )
473                 LDI     3,BK                    ; initialize Block-size to 3
474                 LDI     4,IR0
475                 LDI     4,IR1
476                 LDI     0,RC
477
478                 MPYF3   *AR0,*AR1,R0            ; a2(0)*d(0,n-2) -> R0
479                 MPYF3   *AR0++(1),*AR1--(1)%,R1 ; b2(0)*d(0,n-2) -> R1
480
481               RPTBD   LOOPB
482
483                 MPYF3   *++AR0(1),*AR1,R0       ;a1(0)*d(0,n-1) -> R0
484 ||              ADDF    R0,R2,R2                ;First sum term of d(0,n)
485
486                 MPYF3   *++AR0(1),*AR1--(1)%,R0 ;b1(0)*d(0,n-1) -> R0
487 ||              ADDF3   R0,R2,R2                ;Second sum term of d(0,n)
488                 MPYF3   *++AR0(1),R2,R2         ;b0(0)*d(0,n) -> R2
489 ||              STF     R2,*AR1--(1)%           ;store d(0,n) point to d(0,n-2)
490
491
492                                                 ; loop starts here
493                 MPYF3   *++AR0(1),*++AR1(IR0),R0        ; a2(i)*d(i,n-2) ->R0
494 ||              ADDF3   R0,R2,R2                ; First sum term of y(i-1,n)
495                                                 ; Pipeline hit on previous
496                                                 ; instruction
```

```
497
498                 MPYF3   *++AR0(1),*AR1--(1)%,R1         ; b2(i)*D(i,n-2) -> R1
499 ||              ADDF3   R1,R2,R2                        ; Second sum term of y(i-1,n)
500                 MPYF3   *++AR0(1),*AR1,R0               ; a1(i)*d(i,n-1) -> R0
501 ||              ADDF3   R0,R2,R2                        ; First sum term of d(i,n)
502
503                 MPYF3   *++AR0(1),*AR1--(1)%,R0         ; b1(i)*d(i,n-1) ->R0
504 ||              ADDF3   R0,R2,R2                        ; Second sum term of d(i,n)
505
506 LOOPB           MPYF3   *++AR0(1),R2,R2                 ; b0(i)*d(i,n) ->R2
507 ||              STF     R2,*AR1--(1)%                   ; store d(i,n) point to d(i,n-2)
508
509                                                       ; final summation
510
511                 ADDF    R0,R2                   ; first sum term of y(n-1,n)
512                 ADDF3   R1,R2,R0                ; second sum term of y(n-1,n)
513                 LDI     *AR1--(IR1),R1          ; return to first biquad
514                 LDI     *AR1--(1)%,R2           ; point to d(0,n-1)
515
516
517               STI     AR1,@C_DELAYB           ; store updated delay pointer
518
519
520
521 ********************************************************************************
522 *  end of filtering for CH B                                                   *
523 ********************************************************************************
524
525                 POP     RC              ; recall important registers that were
526                 POP     IR1             ; saved before the filtering
527                 POP     IR0
528                 POP     BK
529                 POP     AR1
530                 POP     AR0
531
532                 LDF     @GAIN_CON,R2            ; load gain correction constant
533                 MPYF    R2,R0                   ; apply gain correction
534                 STF     R0,@CHB_C               ; store current c-filtered ch B sample
535
536 ************************************************
537 * Check to see if we are at the end of a micro blk *
538 ************************************************
539
540
541                 LDI     @DEC_COUNT,R1           ; load decimation counter
542                 LDI     0,R2
543                 CMPI    R1,R2                   ; compare dec_count to zero
544                 BZ      GET_BLR                 ; if result of compare is zero
545                                                  ; then its time to get the blast
546                                                  ; recogn. result from the comm port
547
548 ********************************************************************************
549 *  Compute sum of squares, find max level [NOT END OF MICRO_BLOCK]           *
550 ********************************************************************************
551
552                                                 ; do cha
553                 FLOAT   R10,R10                 ; convert cha data to float pt
554                 MPYF3   R10,R10,R3              ; compute square of cha data->R3
555                 LDF     @CHA_C,R4               ; load c-weighted cha data
556                 MPYF    R4,R4                   ; compute square of c-weighted cha data
557                 LDF     @SQA_MICRO_BLK,R5       ; load the sum-square value for current
558                                                 ; micro block, cha (24 48kHz samples)
559                 LDF     @SQAC_MICRO_BLK,R6      ; load sum-square value for micro
560                                                 ; block for cha, c-weighted
561                 ADDF    R3,R5                   ; add in new sumsquare to micro_blk
562                 ADDF    R4,R6                   ; add in new sumsquare to micro_blk
563                 STF     R5,@SQA_MICRO_BLK       ; store new cha micro_blk
564                 STF     R6,@SQAC_MICRO_BLK      ; store new cha c-weight micro_blk
565
566                                                 ; now do chb
567                 FLOAT   R11,R11                 ; convert chb data to float pt
568                 MPYF3   R11,R11,R7              ; compute square of chb data->R7
569                 LDF     @CHB_C,R8               ; load chb c-weighted data
```

```
570                 MPYF    R8,R8                   ; square chb c-wght data
571                 LDF     @SQB_MICRO_BLK,R5       ; load sum-square for chb micro_blk
572                 LDF     @SQBC_MICRO_BLK,R6      ; load sum-square for chb c-wght m_blk
573                 ADDF    R7,R5                   ; add in new sumsquare to micro_blk
574                 ADDF    R8,R6                   ; add in new sumsquare to micro_blk
575                 STF     R5,@SQB_MICRO_BLK       ; store new chb micro_blk
576                 STF     R6,@SQBC_MICRO_BLK      ; store new chb c-wght micro_blk
577
578                                                 ; now do for cross mult of channels
579                 MPYF3   R10,R11,R7              ; cha*chb->R7
580                 LDF     @SQAB_MICRO_BLK,R5      ; load sum-square for crossmult m_blk
581                 ADDF    R7,R5                   ; add in new sumsquare to micro_blk
582                 STF     R5,@SQAB_MICRO_BLK      ; store updated sumsquare
583                 LDF     @CHA_C,R8               ; load cha c-weighted data
584                 LDF     @CHB_C,R6               ; load chb c-weighted data
585                 MPYF    R8,R6                   ; R6=cha_c*chb_c
586                 LDF     @SQABC_MICRO_BLK,R8     ; load sum-square for crossmult c-weigt
587                 ADDF    R6,R8                   ; add in new sumsquare to micro_blk
588                 STF     R8,@SQABC_MICRO_BLK     ; store new crossmult c-wght micro_blk
589
590                ; check for new max
591
592                 ABSF    R10,R3                  ; abs val of cha data
593                 LDF     @MICRO_MAX_A,R1         ; load max in this micro_blk for cha
594                 LDF     @MICRO_MAX_AC,R2        ; load max in this micro_blk cha c-wght
595                 CMPF    R1,R3                   ; compare cha to max cha in micro blk
596                 BN      GO_ON1                  ; if max in micro is bigger than cha then go on
597                 STF     R3,@MICRO_MAX_A         ; store new found micro max cha
598 GO_ON1          LDF     @CHA_C,R1               ; load cha c-wght
599                 ABSF    R1,R1                   ; absolute val
600                 CMPF    R2,R1                   ; compare cha© to max cha © in micro so far
601                 BN      GO_ON2                  ; if max a© in micro is bigger then go on
602                 STF     R1,@MICRO_MAX_AC        ; store new found micro max cha ©
603 GO_ON2          ABSF    R11,R2                  ; abs val of chb data
604                 LDF     @MICRO_MAX_B,R3         ; load max in this micro_blk for chb
605                 LDF     @MICRO_MAX_BC,R4        ; load max in this micro_blk for chb c-wght
606                 CMPF    R3,R2                   ; compare cha to max thus far
607                 BN      GO_ON3                  ; if max b is bigger then go on
608                 STF     R2,@MICRO_MAX_B         ; store new found max chb
609 GO_ON3          LDF     @CHB_C,R3               ; load chb c-wght
610                 ABSF    R3,R3                   ; abs val chb c-wght
611                 CMPF    R4,R3                   ; compare chb c-wght to max thus far
612                 BN      GO_ON4                  ; if max b © is bigger then go on
613                 STF     R3,@MICRO_MAX_BC        ; store new found max chb c-wght
614 GO_ON4          LDF     @MICRO_MAX_AB,R3        ; load cha*chb max so far in this m_blk
615                 ABSF    R7,R7                   ; take abs val of current cha*chb
616                 CMPF    R3,R7                   ; compare cha*chb to max thus far
617                 BN      GO_ON13                 ; if max old cha*chb is bigger then go on
618                 STF     R7,@MICRO_MAX_AB        ; store new found max cha*chb
619 GO_ON13         LDF     @MICRO_MAX_ABC,R3       ; load cha*chb max so far, c-wght
620                 ABSF    R6,R6                   ; take abs val of current cha*chb c-wght
621                 CMPF    R3,R6                   ; compare cha*chb to max thus far
622                 BN      GO_ON14                ; if old max cha*chb c-wght is bigger then go on
623                 STF     R6,@MICRO_MAX_ABC       ; store new found max cha*chb c-wght
624 GO_ON14         BR      END_OF_ISR              ; jump down to end of service routine
625
626 ********************************************************************************
627 *  done sumsq computations for [NOT END OF MICRO_BLOCK]                        *
628 ********************************************************************************
629
630 ********************************************************************************
631 * start sum of squares, max and blast/ no blast for [END OF MICRO_BLOCK]       *
632 ********************************************************************************
633
634 ************************************
635 * get Blast recogn info             *
636 ************************************
637
638 GET_BLR         LDI     @FIRST_TIME,R8          ; load first time flag. If=1 then its the first
time
639                                                 ; reading from the comm port and there won't be
640                                                 ; anything to read
```

```
641                 CMPI    0,R8
642                 BZ      CONT2                   ; if not first time go on and read from port
643                 STI     0,@FIRST_TIME           ; set flag to NOT first time
644                 LDI     0,R9                    ; load a zero into BLR flag
645                 BR      CONT3
646 CONT2           LDI     *AR5,R9                 ; load blast recogn. flag from CPU_B
647                                                  ; flag=1 ->blast
648                                                  ; flag=0 ->no blast
649 CONT3           STI     R9,@BLR_FLAG            ; save current blr flag
650                 LDI     @TSB_NUM_BLR,R8         ; load number of blr's in tenth sec blk
651                 ADDI    R9,R8                   ; add current flag to total
652                 STI     R8,@TSB_NUM_BLR         ; store updated total
653
654 ************************************************
655 * finish off sum of squares for this micro_block *
656 ************************************************
657
658                 MPYF3   R10,R10,R3              ; compute square of cha data->R3
659                 LDF     @CHA_C,R4               ; load c-weighted cha data
660                 MPYF    R4,R4                   ; compute square of c-weighted cha data
661                 LDF     @SQA_MICRO_BLK,R5       ; load the sum-square value for current
662                                                 ; micro block, cha (24 48kHz samples)
663                 LDF     @SQAC_MICRO_BLK,R6      ; load sum-square value for micro
664                                                 ; block for cha, c-weighted
665                 ADDF    R3,R5                   ; add in new sumsquare to micro_blk
666                 ADDF    R4,R6                   ; add in new sumsquare to micro_blk
667                 STF     R5,@SQA_MICRO_BLK       ; store new cha micro_blk
668                 STF     R6,@SQAC_MICRO_BLK      ; store new cha c-weight micro_blk
669
670                                                 ; now do ch b
671                 MPYF3   R11,R11,R7              ; compute square of chb data->R7
672                 LDF     @CHB_C,R8               ; load chb c-weighted data
673                 MPYF    R8,R8                   ; square chb c-wght data
674                 LDF     @SQB_MICRO_BLK,R5       ; load sum-square for chb micro_blk
675                 LDF     @SQBC_MICRO_BLK,R6      ; load sum-square for chb c-wght m_blk
676                 ADDF    R7,R5                   ; add in new sumsquare to micro_blk
677                 ADDF    R8,R6                   ; add in new sumsquare to micro_blk
678                 STF     R5,@SQB_MICRO_BLK       ; store new chb micro_blk
679                 STF     R6,@SQBC_MICRO_BLK      ; store new chb c-wght micro_blk
680
681                                                 ; now do crossmult
682                 LDF     @CHA_C,R3               ; load cha c-wght
683                 LDF     @CHB_C,R4               ; load chb c-wght
684                 MPYF3   R10,R11,R7              ; R7=cha*chb
685                 MPYF3   R3,R4,R8                ; R8=cha_c*chb_c
686                 LDF     @SQAB_MICRO_BLK,R5      ; load sumsquare for cha*chb m_blk
687                 LDF     @SQABC_MICRO_BLK,R6     ; load sumsquare for cha*chb c-wght m_blk
688                 ADDF    R7,R5                   ; add in new sumsquare
689                 ADDF    R8,R6                   ; add in new sumsquare c-wght
690                 STF     R5,@SQAB_MICRO_BLK      ; store new cha*chb sumsq
691                 STF     R6,@SQABC_MICRO_BLK     ; store new cha_c*chb_c sumsq
692
693
694 *******************************************
695 * find final max vals for micro block       *
696 *******************************************
697
698                 ABSF    R10,R3                  ; abs val of cha data
699                 LDF     @MICRO_MAX_A,R1         ; load max in this micro_blk for cha
700                 LDF     @MICRO_MAX_AC,R2        ; load max in this micro_blk cha c-wght
701                 CMPF    R1,R3                   ; compare cha to max cha in micro
702                 BN      GO_ON5                  ; if max in micro is bigger than cha then go on
703                 STF     R3,@MICRO_MAX_A         ; store new found micro max cha
704 GO_ON5          LDF     @CHA_C,R1               ; load cha c-wght
705                 ABSF    R1,R1                   ; absolute val
706                 CMPF    R2,R1                   ; compare cha© to max cha © in micro so far
707                 BN      GO_ON6                  ; if max a© in micro is bigger then go on
708                 STF     R1,@MICRO_MAX_AC        ; store new found micro max cha ©
709 GO_ON6          ABSF    R11,R2                  ; abs val of chb data
710                 LDF     @MICRO_MAX_B,R3         ; load max in this micro_blk for chb
711                 LDF     @MICRO_MAX_BC,R4        ; load max in this micro_blk for chb c-wght
712                 CMPF    R3,R2                   ; compare cha to max thus far
713                 BN      GO_ON7                  ; if max b is bigger then go on
```

```
714                 STF     R2,@MICRO_MAX_B         ; store new found max chb
715 GO_ON7          LDF     @CHB_C,R3               ; load chb c-wght
716                 ABSF    R3,R3                   ; abs val chb c-wght
717                 CMPF    R4,R3                   ; compare chb c-wght to max thus far
718                 BN      GO_ON8                  ; if max b © is bigger then go on
719                 STF     R3,@MICRO_MAX_BC        ; store new found max chb c-wght
720 GO_ON8          LDF     @MICRO_MAX_AB,R3        ; load old max for cha*chb
721                 ABSF    R7,R7                   ; take abs of current cha*chb
722                 CMPF    R3,R7                   ; compare cha*chb to max thus far
723                 BN      GO_ON15                 ; if old max cha*chb is bigger then go on
724                 STF     R7,@MICRO_MAX_AB        ; store new found max cha*chb
725 GO_ON15         LDF     @MICRO_MAX_ABC,R3       ; load old max for cha_c*chb_c
726                 ABSF    R8,R8                   ; take abs val of current cha_c*chb_c
727                 CMPF    R3,R8                   ; compare cha_c*chb_c to max thus far
728                 BN      GO_ON16                 ; if old max cha_c*chb_c is bigger then go on
729                 STF     R8,@MICRO_MAX_ABC       ; store new found max cha_c*chb_c
730 GO_ON16         LDI     @MICRO_BLK_CNT,R1       ; load micro block counter
731
732
733 ********************************************
735 * add micro sum of squares info to TSB info   *
736 ********************************************
737
738                 LDF     @SQA_MICRO_BLK,R2       ; load micro sum of squares for cha
739                 LDF     @SQA_TSB,R3             ; load TSB sum of squares for cha
740                 ADDF    R2,R3                   ; add in micro sumsq to TSB sumsq cha
741                 LDF     @SQAC_MICRO_BLK,R4      ; load micro sumsq for cha c-wght
742                 LDF     @SQAC_TSB,R5            ; load TSB sumsq for cha c-wght
743                 ADDF    R4,R5                   ; add in micro sumsq to TSB sumsq cha©
744                 STF     R3,@SQA_TSB             ; store TSB sumsq for cha
745                 LDF     @SQB_MICRO_BLK,R2       ; load micro sumsq for chb
746                 LDF     @SQB_TSB,R3             ; load TSB sumsq for chb
747                 ADDF    R2,R3                   ; add in micro sumsq to TSB sumsq chb
748                 STF     R5,@SQAC_TSB            ; store TSB sumsq for cha c-wght
749                 LDF     @SQBC_MICRO_BLK,R4      ; load micro sumsq for chb c-wght
750                 LDF     @SQBC_TSB,R5            ; load TSB sumsq for chb c-wght
751                 ADDF    R4,R5                   ; add in micro sumsq to TSB sumsq chb©
752                 STF     R3,@SQB_TSB             ; store TSB sumsq for chb
753                 STF     R5,@SQBC_TSB            ; store TSB sumsq for chb©
754                 LDF     @SQAB_MICRO_BLK,R2      ; load micro blk sum for cha*chb
755                 LDF     @SQAB_TSB,R3            ; load tsb sum of squares for cha*chb
756                 ADDF    R2,R3                   ; add in micro sumsq for cha*chb
757                 STF     R3,@SQAB_TSB            ; store  updated tsb sumsq
758                 LDF     @SQABC_MICRO_BLK,R4     ; load micro blk sum for cha*chb c-wght
759                 LDF     @SQABC_TSB,R5           ; load tsb sum for cha*chb c-wght
760                 ADDF    R4,R5                   ; add in micro sumsq for cha*chb c-wght
761                 STF     R5,@SQABC_TSB           ; store updated tsb sumsq
762
763
764
765 ********************************************
766 * check to see if current micro blk contains  *
767 * the max value in TSB thus far               *
768 ********************************************
769                 LDI     @MICRO_BLK_CNT,R8       ; load micro_blk counter into R8
770                 LDF     @TSB_MAX_A,R1           ; load TSB max for cha
771                 LDF     @MICRO_MAX_A,R2         ; load micro max for cha
772                 CMPF    R1,R2                   ; compare the two max vals
773                 BN      GO_ON9                  ; if TSB max is greater then go on
774                 STF     R2,@TSB_MAX_A           ; store new found max
775                 STI     R8,@TSB_MAX_POS_A       ; save max position=micro block counter
776 GO_ON9          LDF     @TSB_MAX_AC,R2          ; load TSB max for cha ©
777                 LDF     @MICRO_MAX_AC,R3        ; load micro max for cha ©
778                 CMPF    R2,R3                   ; compare the two max vals
779                 BN      GO_ON10                 ; if TSB max is greater then go on
780                 STF     R3,@TSB_MAX_AC          ; store new found max
781                 STI     R8,@TSB_MAX_POS_AC      ; store max position
782 GO_ON10         LDF     @TSB_MAX_B,R2           ; load TSB max for chb
783                 LDF     @MICRO_MAX_B,R3         ; load micro max for chb
784                 CMPF    R2,R3                   ; compare the two max vals
785                 BN      GO_ON11                 ; if TSB max is greater then go on
786                 STF     R3,@TSB_MAX_B           ; store new found max
```

```
787                 STI     R8,@TSB_MAX_POS_B       ; store max position
788 GO_ON11         LDF     @TSB_MAX_BC,R2          ; load TSB max for chb©
789                 LDF     @MICRO_MAX_BC,R3        ; load micro max for chb©
790                 CMPF    R2,R3                   ; compare the two max vals
791                 BN      GO_ON12                 ; if TSB max is greater then go on
792                 STF     R3,@TSB_MAX_BC          ; store new found max
793                 STI     R8,@TSB_MAX_POS_BC      ; store max position
794 GO_ON12         LDF     @TSB_MAX_AB,R2          ; load tsb max for cha*chb
795                 LDF     @MICRO_MAX_AB,R3        ; load micro max for cha*chb
796                 CMPF    R2,R3                   ; compare the two max vals
797                 BN      GO_ON17                 ; if tsb max is greater then go on
798                 STF     R3,@TSB_MAX_AB          ; store new found tsb max
799                 STI     R8,@TSB_MAX_POS_AB      ; store max position
800 GO_ON17         LDF     @TSB_MAX_ABC,R2         ; load tsb max for cha_c*chb_c
801                 LDF     @MICRO_MAX_ABC,R3       ; load micro max for   “
802                 CMPF    R2,R3                   ; compare the two max vals
803                 BN      GO_ON18                 ; if tsb max is bigger then go on
804                 STF     R3,@TSB_MAX_ABC         ; store new found max
805                 STI     R8,@TSB_MAX_POS_ABC     ; store max position
806 GO_ON18         LDI     199,R2                  ; load max for micro counter
807
808
809 *******************************************
810 * clear micro blk variables               *
811 *******************************************
812
813                 LDF     0,R3                    ; load floating pt zero into all micro vars
814                 STF     R3,@SQA_MICRO_BLK
815                 STF     R3,@SQAC_MICRO_BLK
816                 STF     R3,@SQB_MICRO_BLK
817                 STF     R3,@SQBC_MICRO_BLK
818                 STF     R3,@MICRO_MAX_A
819                 STF     R3,@MICRO_MAX_AC
820                 STF     R3,@MICRO_MAX_B
821                 STF     R3,@MICRO_MAX_BC
822                 STF     R3,@SQAB_MICRO_BLK
823                 STF     R3,@SQABC_MICRO_BLK
824                 STF     R3,@MICRO_MAX_AB
825                 STF     R3,@MICRO_MAX_ABC
826
827

828 *******************************************
829 * check to see if current micro block ends    *
830 * a tenth second block (MICRO_BLK_CNT=199)    *
831 *******************************************
832
833                 LDI     @MICRO_BLK_CNT,R1       ; load micro_blk counter
834                 CMPI    R1,R2                   ; compare micro block counter to 199
835                 BZ      END_OF_TSB              ; if counter=199 then goto end TSB calcs
836                 ADDI    1,R1                    ; increment micro counter
837                 STI     R1,@MICRO_BLK_CNT       ; store updated counter
838                 BR      END_OF_ISR              ; else goto end of service routine
839
840 *************************************************************************
841 *   Close out a Tenth Second Block (TSB):                                 *
842 *    All data in tsb_etc locations should be correct. Now write it out to  *
843 *    the  DPRAM circular buffer(length 950 (50blks*19words))               *
844 *    Current starting address of data in stored in a0000400h in DPRAM.     *
845 *************************************************************************
846
847 END_OF_TSB      LDI     950,BK                 ; load size of DPRAM circ. buff. (5secs*19words)
848                 PUSH    AR5                     ; save contents of AR5 on stack
849                 PUSH    AR6                     ; save contents of AR6 on stack
850                 LDA     @CURRENT_ADDR,AR6       ; load addr of spot in DPRAM where current AR7
851         ; is stored so PC knows where most current data
852         ; lies
853                 STI     AR7,*AR6                ; store contents of AR7 to current addr
854
855                 LDA     @SEND_DATA_ADDR,AR5     ; load first address of data to send to DPRAM
856                 LDF     *AR5++,R1               ; load SQA_TSB
857                 TOIEEE  R1,R1                   ; convert SQA_TSB to IEEE float
```

```
858                 LDF     *AR5++,R2               ; load SQAC_TSB
859 ||              STF     R1,*AR7++%              ; store SQA_TSB in circ buff
860                 TOIEEE  R2,R2                   ; convert SQAC_TSB to IEEE float
861                 LDF     *AR5++,R3               ; load SQB_TSB
862 ||              STF     R2,*AR7++%              ; store SQAC_TSB
863                 TOIEEE  R3,R3                   ; convert SQB_TSB to IEEE float
864                 LDF     *AR5++,R4               ; load SQBC_TSB
865 ||              STF     R3,*AR7++%              ; store SQB_TSB
866                 TOIEEE  R4,R4                   ; convert SQBC_TSB to IEEE float
867                 LDF     *AR5++,R5               ; load TSB_MAX_A
868 ||              STF     R4,*AR7++%              ; store SQBC_TSB
869                 TOIEEE  R5,R5                   ; convert TSB_MAX_A to IEEE float
870                 LDF     *AR5++,R6               ; load TSB_MAX_AC
871 ||              STF     R5,*AR7++%              ; store TSB_MAX_A
872                 TOIEEE  R6,R6                   ; convert TSB_MAX_AC to IEEE float
873                 LDF     *AR5++,R7               ; load TSB_MAX_B
874 ||              STF     R6,*AR7++%              ; store TSB_MAX_AC
875                 TOIEEE  R7,R7                   ; convert TSB_MAX_B to IEEE float
876                 LDF     *AR5++,R1               ; load TSB_MAX_BC
877 ||              STF     R7,*AR7++%              ; store TSB_MAX_B
878                 TOIEEE  R1,R1                   ; convert TSB_MAX_BC to IEEE float
879                 LDI     *AR5++,R2               ; load TSB_MAX_POS_A
880                 STF     R1,*AR7++%              ; store TSB_MAX_BC
881                 LDI     *AR5++,R3               ; load TSB_MAX_POS_AC
882 ||              STI     R2,*AR7++%              ; store TSB_MAX_POS_A
883                 LDI     *AR5++,R4               ; load TSB_MAX_POS_B
884 ||              STI     R3,*AR7++%              ; store TSB_MAX_POS_AC
885                 LDI     *AR5++,R5               ; load TSB_MAX_POS_BC
886 ||              STI     R4,*AR7++%              ; store TSB_MAX_POS_B
887                 LDI     *AR5++,R6               ; load TSB_NUM_BLR
888 ||              STI     R5,*AR7++%              ; store TSB_MAX_POS_BC
889                 STI     R6,*AR7++%              ; store TSB_NUM_BLR
890                 LDF     *AR5++,R2               ; load SQAB_TSB
891                 TOIEEE  R2,R2                   ; convert to IEEE float
892                 LDF     *AR5++,R3               ; load SQABC_TSB
893 ||              STF     R2,*AR7++%              ; store SQAB_TSB
894                 TOIEEE  R3,R3                   ; convert to IEEE float
895                 LDF     *AR5++,R4               ; load TSB_MAX_AB
896 ||              STF     R3,*AR7++%              ; store SQABC_TSB
897                 TOIEEE  R4,R4                   ; convert to IEEE float
898                 LDF     *AR5++,R5               ; load TSB_MAX_ABC
899 ||              STF     R4,*AR7++%              ; store TSB_MAX_AB
900                 TOIEEE  R5,R5                   ; convert to IEEE float
901                 LDI     *AR5++,R6               ; load TSB_MAX_POS_AB
902                 STF     R5,*AR7++%              ; store TSB_MAX_ABC
903                 LDI     *AR5,R2                 ; load TSB_MAX_POS_ABC
904 ||              STI     R6,*AR7++%              ; store TSB_MAX_POS_AB
905                 STI     R2,*AR7++%              ; store TSB_MAX_POS_ABC
906                 POP     AR6                     ; restore AR6
907                 POP     AR5                     ; restore AR5
908                 LDI     0,R6
909                 STI     R6,@MICRO_BLK_CNT       ; zero the micro blk counter
910
911 ******************************************
912 * clear all the TSB variables            *
913 ******************************************
914
915                 LDF     0,R1
916                 STF     R1,@SQA_TSB
917                 STF     R1,@SQAC_TSB
918                 STF     R1,@SQB_TSB
919                 STF     R1,@SQBC_TSB
920                 STF     R1,@SQAB_TSB
921                 STF     R1,@SQABC_TSB
922                 STF     R1,@TSB_MAX_A
923                 STF     R1,@TSB_MAX_AC
924                 STF     R1,@TSB_MAX_B
925                 STF     R1,@TSB_MAX_BC
926                 STF     R1,@TSB_MAX_AB
927                 STF     R1,@TSB_MAX_ABC
928                 LDI     0,R1
929                 STI     R1,@TSB_MAX_POS_A
930                 STI     R1,@TSB_MAX_POS_AC
```

```
931                 STI     R1,@TSB_MAX_POS_B
932                 STI     R1,@TSB_MAX_POS_BC
933                 STI     R1,@TSB_MAX_POS_AB
934                 STI     R1,@TSB_MAX_POS_ABC
935                 STI     R1,@TSB_NUM_BLR
936
937 ************************************************************
938 *  end of the service routine proper                       *
939 ************************************************************
940
941
942 END_OF_ISR      LDI     @CHA_INTEGER,R1         ; load cha data
943                 STI     R1,*AR1                 ; output cha data to cha output
944                                                 ;   on D/A converter
945
946                 LDI     @BLR_FLAG,R9
947                 LSH     14,R9                   ; shift R9=blr flag
948                 STI     R9,*AR3                 ; output blr flag to chb output
949                                                 ;   on D/A converter
950
951
952                 RETI
953
954
955 ********************************************************************************
956 *  END OF PROGRAM                                                              *
957 ********************************************************************************
958
959                 .end
```

Appendix D3: C-Weighting Filter

The ideal analog C-weighting has the frequency response given by the transfer function

$$H(s) = \frac{K_1 s^2}{(s+\omega_1)^2 (s+\omega_2)^2}$$

where $\omega_1=2\pi$ 20.598997, $\omega_2=2\pi$12194.22, and K_1 is a constant (as per ANSI S1.4). MATLAB (The Mathworks, Inc., Natick, MA) was used to compute the frequency response of this filter. The constant, K_1, was found to be 5.912387×10^9 by requiring that the response have unity magnitude at 1 kHz. This transfer function was then transformed to the digital domain via the bilinear transform. The nonlinear mapping meant that the high frequency response of the resulting digital transfer function did not match well with the desired response. To correct for this high frequency, poles were altered in an ad hoc fashion until the desired response was achieved.

To implement this IIR digital filter using the TI TMS320C40 DSP processor, the transfer function had to be broken down into 2-second order biquad sections of the form

$$H_1(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

This was done by finding the roots of the transfer function and grouping them in second-order sections. Note that the pairing of poles and zeros in this case was crucial to the stability of the filter. There are two poles at z=1 and two zeros at z=0.997. Because (1) IIR filters are inherently more sensitive to numerical problems and (2) these poles and zeros nearly cancel each other out, these pairs were placed in different biquad sections. If they were placed in the same biquad, the filter became unstable. The final transfer functions for the two biquad sections were:

$$H_0(z) = \frac{0.44519794 + 0.89039588 z^{-1} + 0.44519794 z^{-2}}{1 - 0.2231468 z^{-1} + 0.000122448623 z^{-2}}$$

$$H_1(z) = \frac{0.44519794 - 0.89039588z^{-1} + 0.44519794z^{-2}}{1 - 1.9946068z^{-1} + 0.99461403z^{-2}}$$

A gain correction factor is post multiplied to the filtered data to make sure that the response is unity at 1 kHz as measured from the actual system. This factor was measured to be 1.3101327. The measured response of the digital filter is plotted in Figure D1 with the ideal analog response. The two responses are very close, only differing at the extreme high and low frequencies. The resulting response is well within the tolerances for a Type 0 sound level meter as described in ANSI S1.4-1983.

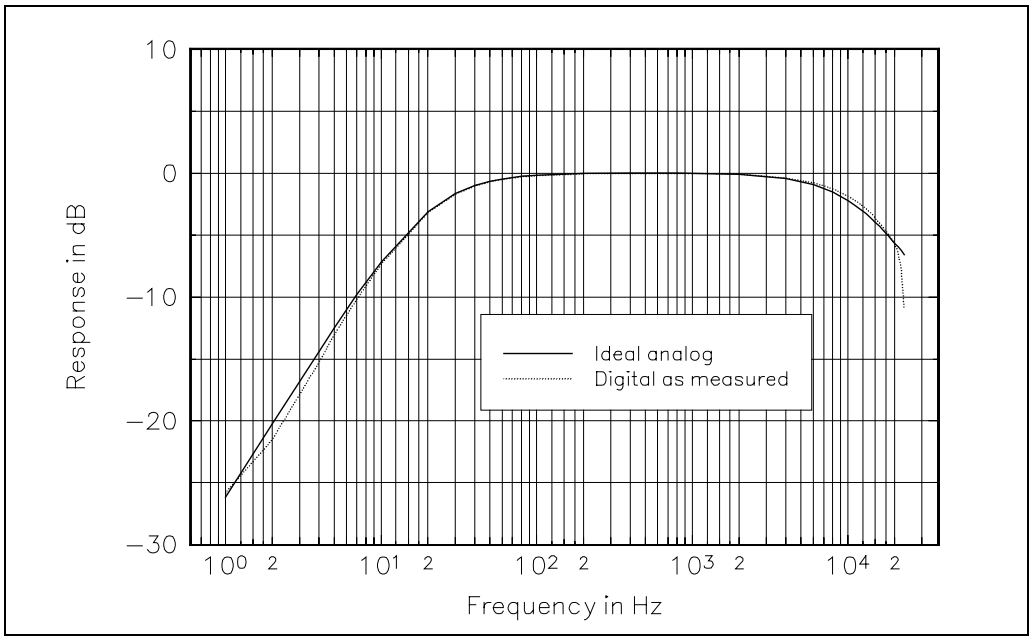


Figure D1. Frequency response of C-filter ideal analog and measured digital response.

It should be noted that the type of digital filter used here (IIR) was chosen because a finite impulse response (FIR) filter could not be designed that could be implemented in real time. It could not be designed because of the high number of coefficients required to match the response of the ideal filter.

Appendix E: Field Unit Schematics

Table E1. Connector pin assignments.

Pin Number	Function
1	Common. Relay Pole 1
2	Normally Closed Contact, Pole 1
3	Normally Open Contact, Pole 1
4	Opto-Isolated Reset Out]
5	Opto-Isolated Reset Source
6	Opto-Isolated NOT Reset Out
7	Opto-Isolated NOT Reset Source
8	Buffered Reset Out
9	+5 VDC Unfused, 1 A max
10	+5 VDC Unfused, 1 A max
11	+5 VDC Unfused, 1 A max
12	+5 VDC Unfused, 1 A max
13	56 kHz square wave while WDOG enabled
14	Common. Relay Pole 2
15	Normally Closed Contact, Pole 2
16	Normally Open Contact, Pole 2
17	Source Opto-Isolated Input #0
18	Return Opto-Isolated Input #0
19	Return Opto-Isolated Input #1
20	Source Opto-Isolated Input #1
21	Tachometer Input from Fan
22	Ground
23	Ground
24	Ground
25	Ground

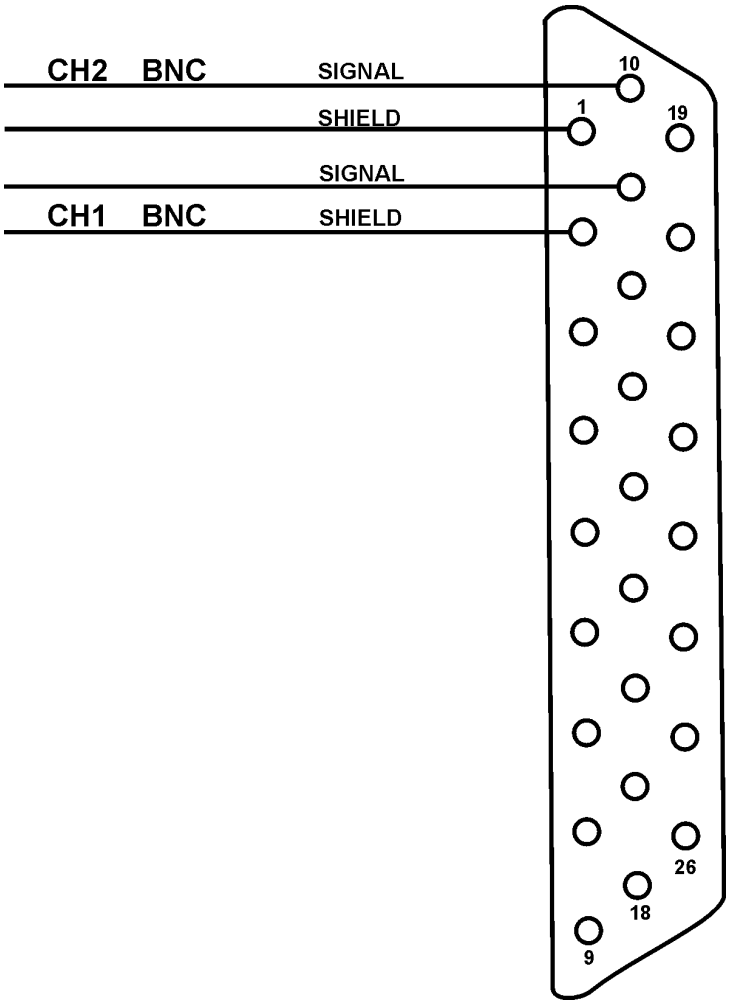


Figure E1. The analog input connector on the DPC-C40 DSP motherboard.

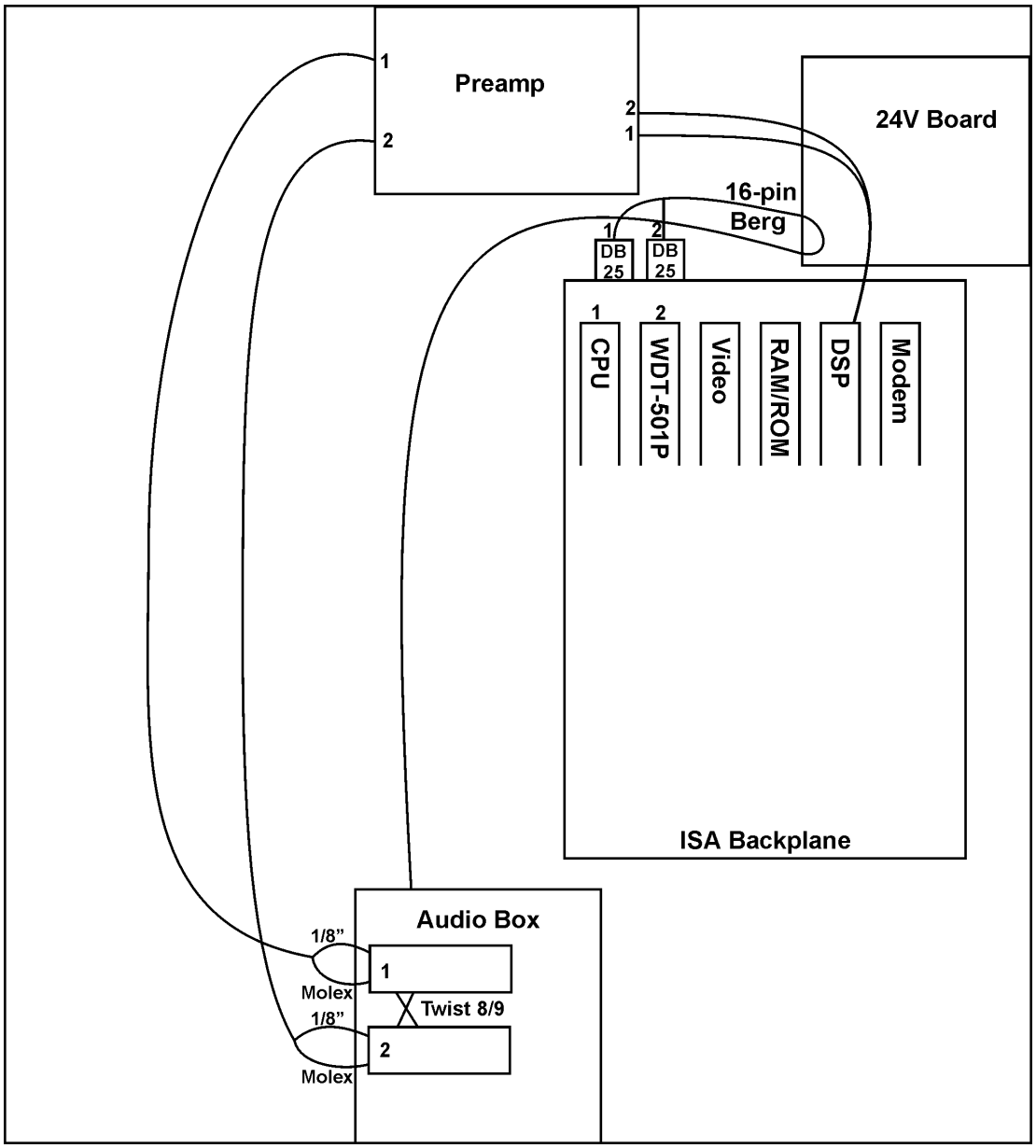
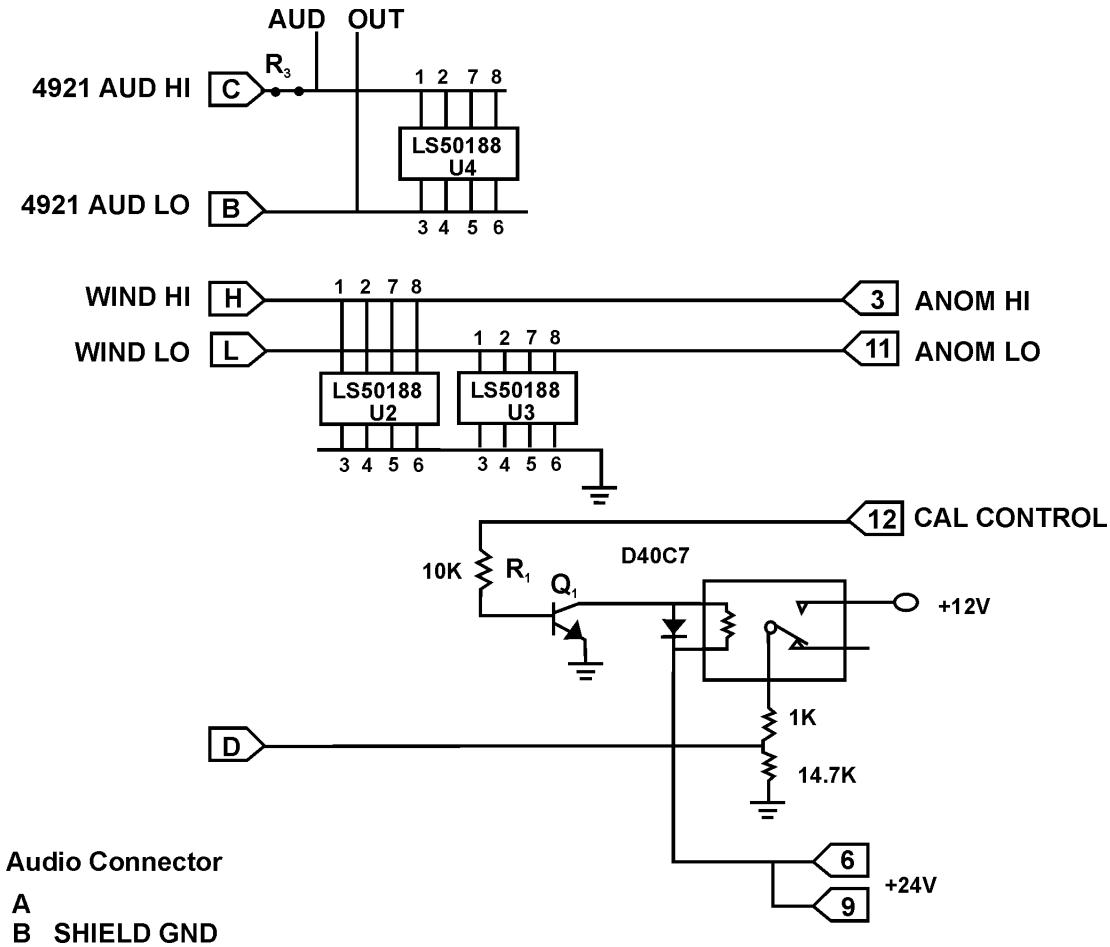


Figure E2. Interconnections between audio input boards, 24-V board, ISA card cage, and microphone power supply/preamp.



Audio Connector

A	
B	SHIELD GND
C	SIGNAL
D	CAL START
E	BIAS
F	
H	WIND HI
J	+9V
K	-9V
L	WIND LO
M	
N	+12V

Figure E3. Audio input PCB schematic and connector pinout.

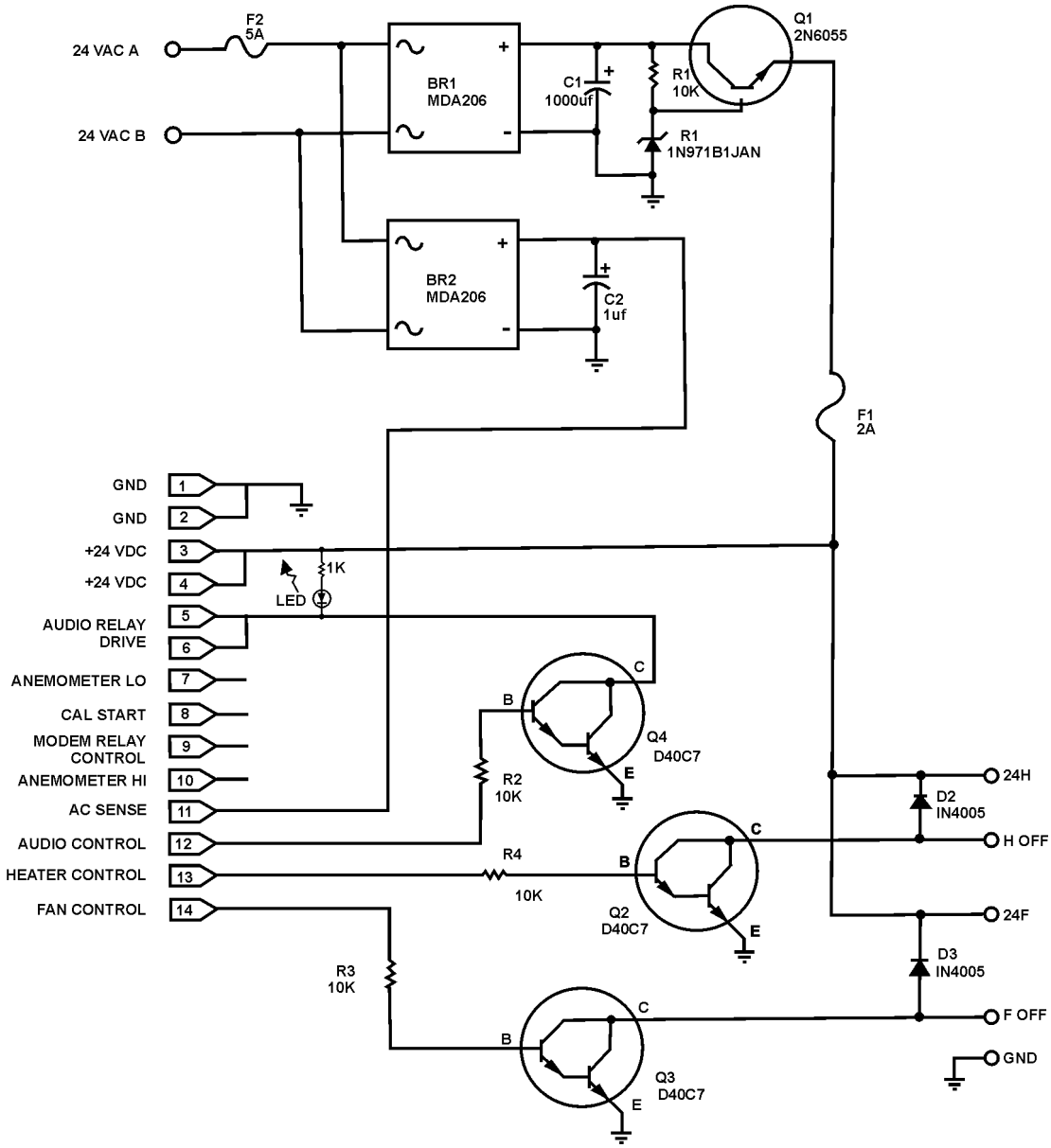


Figure E4. Schematic of 24-V board.

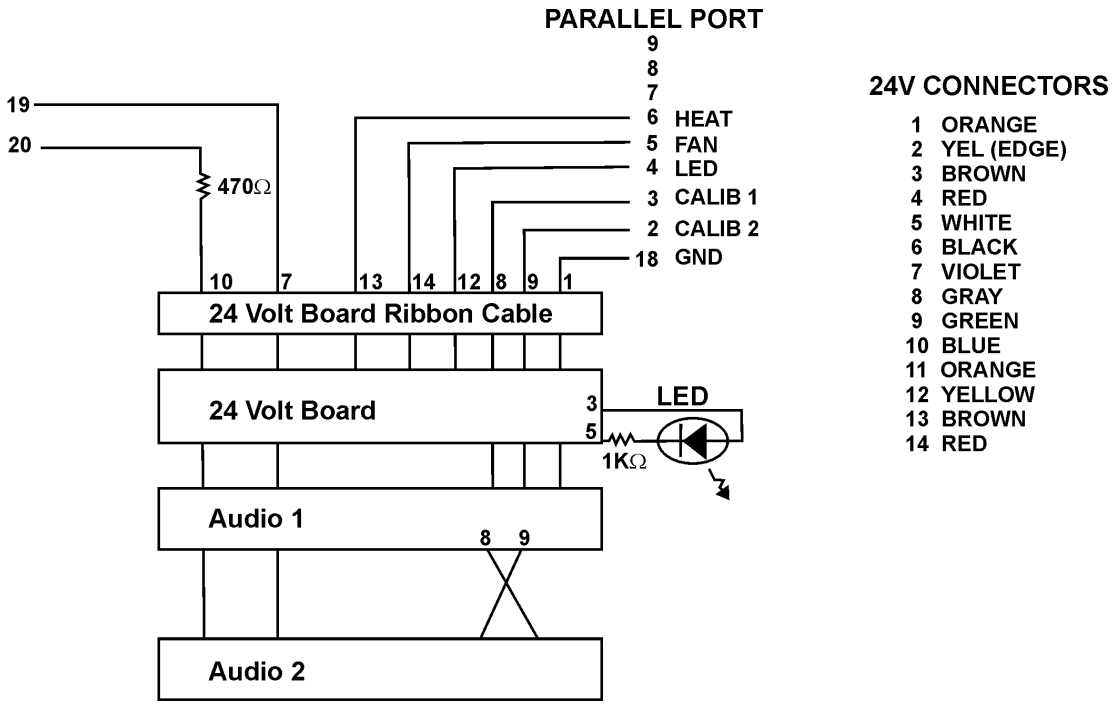


Figure E5. Wiring harness connecting the ISA card cage, the 12-V board, and the audio input boards.

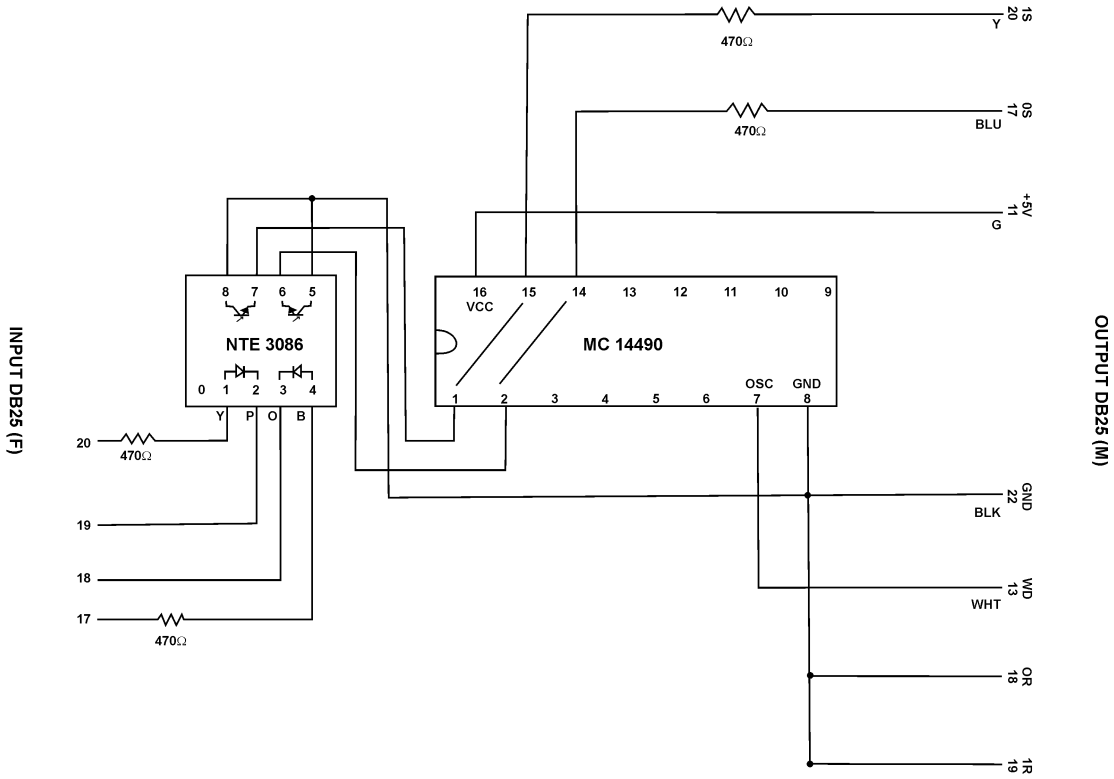


Figure E6. Wind meter debounce circuit schematic, which is inserted between the WDT-501P watchdog card and the wiring harness.

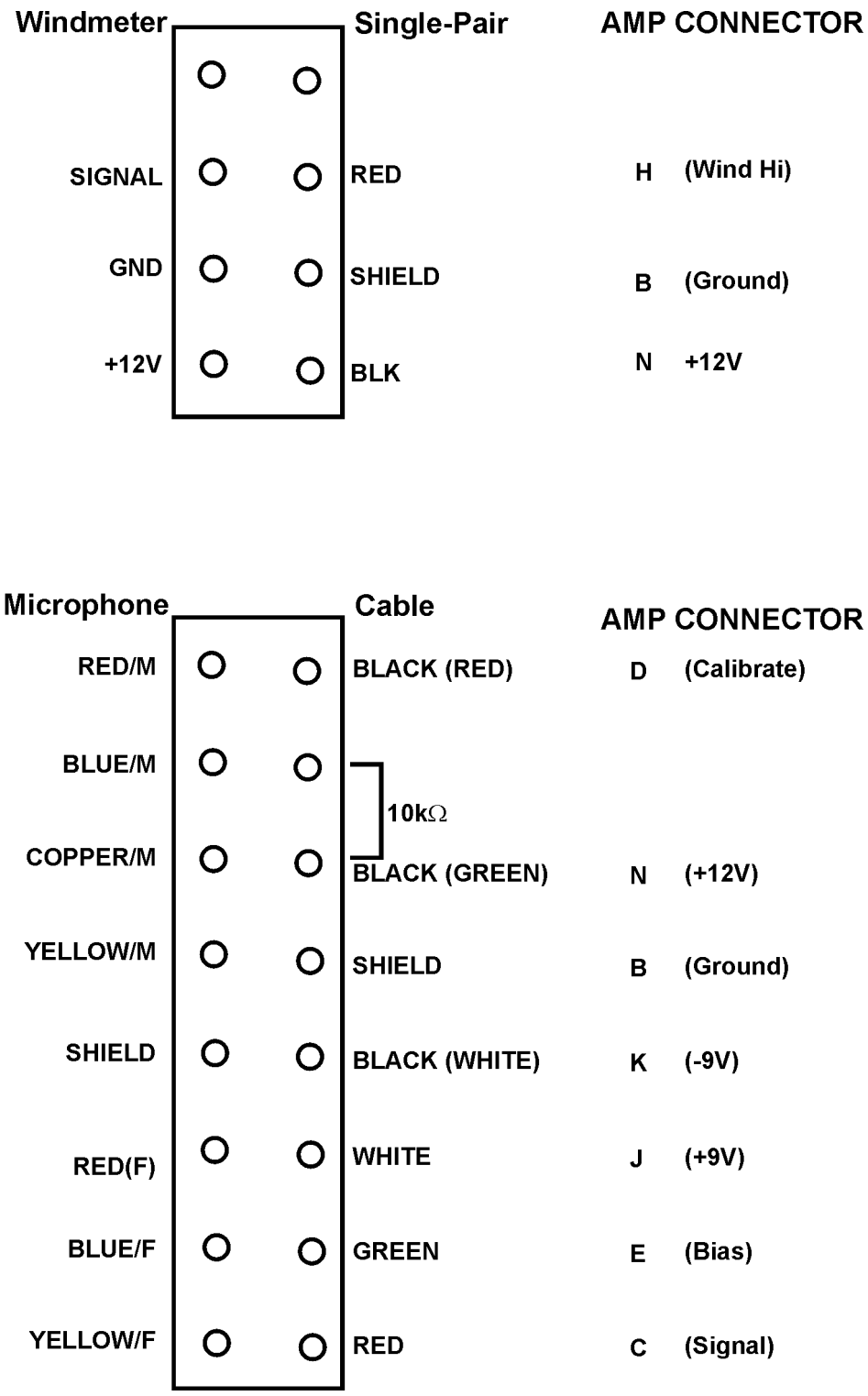


Figure E7. Connections between field unit, microphones, and wind meter. Channel 1 (top) includes the wind meter block; channel 2 (bottom) does not.

Appendix F: Field Unit Communications Protocol Specification

The specification ("Communication Protocol for the CERL Noise Monitoring/Warning System") on the following pages is the protocol for the original CERL field unit. This prototype field unit adheres to this specification, with the following exceptions:

- Support for bit rates greater than 300 bps.
- Flow control is sometimes inexact due to limitations imposed by the 28.8 kbps internal modem's buffering. An incoming ACK or NAK may not immediately terminate an outgoing packet.
- Timings are somewhat inexact.
- An alternate mechanism is provided for deleting all blocks — an attempt to set an autocalibration interval of greater than 10 hours via the mode set command will wipe out all data stored in the monitor (i.e., by sending 0x2B as the autocal interval = 11 hours). This is because, for some reason, the base station code sends this sequence instead of the documented sequence.
- Listen mode is not implemented.
- The meaning of the wind flag has changed.

The wind flag returned as part of Packet Type A and H is the sum of the following:

0x01	Windy block (wind speed exceeded threshold)
0x10	No blast detection
0x20	Did not exceed filter requirements

with 0x20 added onto the result to ensure that the result is a printable ASCII character.

Communication Protocol for the CERL Noise Monitoring/Warning System

L. M. Lendrum
A. J. Averbuch

Revised March 23, 1992

Table of Contents

1) **Hardware Protocol**

1.1) The physical communication between base station and field units is via 300 baud modem (Bell 103). Due to programming consideration outside this protocol, the modem must employ the Hayes "AT" modem command set.

1.2) The communication with the modem is through the serial port of the base station computer (RS-232). The serial ports parameters are 300 baud, 8 data bits, 1 stop bit, no parity. Standard RS-232 flow control signals are supported.

2) **Packet Protocol**

2.1) General

The base station initiates and masters all exchanges using a packet protocol consisting of control characters, and printable ASCII characters only. The field unit may respond to a packet in one of four ways;

a) If the message does not request data, the field unit will respond by transmitting ACK (15h) to the base station if the packet was received correctly and the message was understood.
b) If the message does not request data, the field unit will respond by transmitting NAK (06h) if either the packet was incorrect (garbled in transmission) or the message was not understood (command not valid or format not valid).
c) If the field unit receives nothing, it will, of course, send nothing. In this case, the originator will send the packet again (up to three times).
d) If the message requested data to be transmitted back to the base, the field unit will respond by sending an appropriate packet back to the base. The base must respond with ACK or NAK as appropriate.

2.2) Valid ASCII Characters

The following are valid characters in the protocol. No others are allowed and only three may be used outside of a packet.

a) ACK (06h), NAK (15h), and DLE (10h) are valid characters outside a packet.

ACK means the previous packet has been successfully processed.

NAK means the previous packet was not successfully processed.

DLE is the signal to terminate the communication link.

b) SYN (16h), STX (02h), ETX (03h), and all printable ASCII characters (20h through 7fh) are permitted within a packet.

2.3) Block Numbers

Each packet contains a block number in addition to the message and other components. The purpose of block numbers is to guarantee that packets have not been lost or that packets are not processed twice.

At the beginning of each communication session (modem link established), both the base station and the field unit will expect the first packet received will have a block number equal to 1. For each successive packet, the block number will be incremented.

Block numbers start at 01h and increase up to and including 3Fh. After 3fh, the block numbers wraps around to 0h.

BKN (the character representing the block number) is formed by adding 20h to the block number (in hex). This transforms the block number into a printable ASCII character.

NBK (the character representing the compliment of the block number) is formed by adding the ones compliment of the block number masked by 3fh to 20h.

Example: If the block number is 15 hex, then BKN is 35 hex (ASCII character 5) and NBK is 4A hex (ASCII character J).

2.4) Check Word

A check word is formed for each packet and transmitted as part of the packet. The check word is a 12-bit word formed by starting with zero and XORing each byte in the packet (except SYN characters) into the check word and rotating left one position in a 12-bit word each time.

CKL (the character representing check word low) is formed by adding the low six bits of the check word to 20h.

CKH (the character representing check word high) is formed by adding the high six bits of the check word to 20h.

Example: Assume that the following characters are to be combined into a check word: 02h, 35h,4ah,47h,20h,03h.

Start with zero	0000 0000 0000
XOR with 0000 0010	0000 0000 0010
shift left	0000 0000 0100
XOR with 0011 0101	0000 0011 0001
shift left	0000 0110 0010
XOR with 0100 1010	0000 0010 1000
shift left	0000 0101 0000
XOR with 0100 0111	0000 0001 0111
shift left	0000 0010 1110
XOR with 0010 0000	0000 0000 1110
shift left	0000 0001 1100
XOR with 0000 0011	0000 0001 1111
shift left	0000 0011 1110

CKL is 5Eh (ASCII ^) and CKH is 20h (ASCII SP).

2.5) Packet Format

A packet is defined as:

	three SYN characters
*[1]	STX character
*	BKN character as defined above
*	NBK character as defined above
*	at least one printable data character (20h - 7Fh)
*	ETX character
	CKL character as defined above
	CKH character as defined above

1. The characters marked are those used to form the check word.

Example: Assume the next block number is 15 hex and the command to be send to the monitor is DISABLE DATA ACQUISITION. This command is the letter "G" followed by 00h. However, the message characters must be printable ASCII; we can guarantee this symbol is printable by adding 20 hex to the value (in this case 00 hex).

The packet would consist of the following:

16h 16h 16h 02h 35h 4ah 47h 20h 03h 56h 20h

or

SYN SYN SYN STX 5 J G SP ETX ^ SP

2.6) Flow Control and Error Checking

As mentioned earlier, each packet contains a block number which is incremented after each successful packet transmission.

Suppose the base station sends a packet and does not receive a proper response from the field unit. It is outlined later in the section on the command level the protocol what is expected as the response to the various commands and reply formats. The response may be either a packet sent from the field unit or an ACK to acknowledge proper receipt of the command. A period of 0.5 seconds is allowed for a response. If a negative response is received or 0.5 seconds past with no response, the base station will resend the packet. After ten attempts to transmit with the field unit, the base station will sent DLE and hang-up.

If the field unit sends a packet in response to a command from the base station, the field unit also expects to see an ACK to indicate the base station received the data. The field unit uses the same 0.5 second waiting period for a response, and receiving no response, will retransmit 10 times before giving up, sending a DLE and hanging up.

The block numbers are supposed to be checked so that 1) the field unit does not respond to a retransmitted command inappropriately if it had already responded, and 2) the base station does not assume retransmission of the same data by the field unit represents two separate events.

3) **Command Level**

Each packet transmitted by either the base station or the field unit is a command to the field unit or data sent by the field unit in response to a command from the base station. These commands and data are identified by the first character of the body of the packet and consist of upper case ASCII letters. Additional characters will be required in some cases to further identify the action commanded or in the case of data being sent by the field unit, the actual data in a format with will be described later.

Commands sent by the Base Station:

CMD	Description	Expected Response
A	data request	packet
B	header request	packet
C	link request	packet
D	listen mode request	ACK
E	immediate fixed length block	ACK
F	future fixed length block	ACK
G	enable/disable threshold blocks	ACK

Messages sent by the field unit:

Response	Description	Expected Response
A	threshold event data	all ACK's
B	self-calibration data	
C	no more data available	
D	header data	
E	temperature data	
F	voltage data	
G	link response	
H	fixed length block data	
Z	bad data format at field unit	

The full description of these commands and messages are in the following sections. These sections are organized by describing the format of the packet command sent by the base station followed by the format of all of the possible response packets sent by the field unit.

3.1 Data Request

Command ID Character: A

Description:

This command requests a data block be transmitted to the base station by the field unit. Data blocks may by threshold event data, self-calibration data, temperature data, voltage data, or fixed length block data. These data are stored in order of occurrence in the field unit memory and will be transmitted in the order they were stored. One data request command will result in only one data block being transmitted to the base station. Multiple data requests must be transmitted to retrieve multiple data blocks.

Responses from Field Unit:

3.1.1) Threshold Event Data Block

Format ID Character: A

Format:

A ....Format ID character
unit # ....Field unit ID number (0-63 decimal) plus 20h
SEL Data..... .....SEL data consists of 8 bytes (least significant first); these are converted to 16 nibbles (low nibble first, then high nibble); each nibble is added to 20h to form 16 bytes (all printable ASCII characters)
Peak Data ... .Peak data consists of 2 bytes as above; converted to 4 nibbles and added to 20h to yield 4 bytes (all printable)
Sample length.....
Sample length is 6 bytes; each count represents 50 microseconds; convert to 12 printable ASCII characters as above
Peak Position.....Peak Position is 6 bytes; process the same as sample length. Represents the position of the highest magnitude signal in the event relative to the beginning of the event
Time-of-event
tenths..... .....a BCD nibble added to 20h
seconds.... .....a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
minutes ....same as above
hours..... .....same as above
days ....days is 2 binary bytes (least significant first); convert to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978
wind flag ....20h for not windy, 21h-2fh for windy

3.1.2 Self-calibration Data Block

Format ID Character: B

Format:

B..... ....Format ID character
unit # ....Field unit ID number (0-3fh) plus 20h
RMS Data ... .RMS data consists of 8 bytes (least significant first); these are converted to 16 nibbles (low nibble first, then high nibble); each nibble is added to 20h to form 16 bytes (all printable ASCII characters)
Peak Data ... .Peak data consists of 2 bytes as above; converted to 4 nibbles and added to 20h to yield 4 bytes (all printable)
Time-of-event
tenths..... .....a BCD nibble added to 20h
seconds.... .....a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
minutes ....same as above
hours..... .....same as above
days ....days is 2 binary bytes (least significant first); convert to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978

3.1.3 No More Data Available Block

Format ID Character: C

Format:

C ....Format ID character
unit # ....Field unit ID number (0-3fh) plus 20h

3.1.4 Temperature Data Block

Format ID Character: E

Format:

E..... ....Format ID character
unit # ....Field unit ID number(0-3fh) plus 20h
Time-of-event
tenths..... .....a BCD nibble added to 20h
seconds.... .....a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
minutes ....same as above
hours..... .....same as above
days ....days is 2 binary bytes (least significant first); convert to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978
Status ....This is 1 byte of data which is the same as one reads on the NEC portable terminal using "32H IN .HB". The interpretation of this byte can be found in the operations manual. This data is split into nibbles (least significant nibble first) and added to 20h to form 2 printable ASCII characters.

3.1.5 Voltage Data Block

Format ID Character: F

Format:

F..... ....Format ID character
unit # ....Field unit ID number(0-3fh) plus 20h
Time-of-event
tenths..... .....a BCD nibble added to 20h
seconds.... .....a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
minutes ....same as above
hours..... .....same as above
days ....days is 2 binary bytes (least significant first); convert to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978
Status ....This is 1 byte of data which is the same as one reads on the NEC portable terminal using "32H IN .HB". The interpretation of this byte can be found in the operations manual. This data is split into nibbles (least significant nibble first) and added to 20h to form 2 printable ASCII characters.

3.1.6 Fixed Length Data Block[1]

Format ID Character: H

Format:

The format is identical to the format of an SEL data block except the Format ID Character is H instead of A.

3.1.7 Bad Data Block

Format ID Character: Z

Format:

Z..... .....Format ID character
unit# . .....Field unit ID number(0-63 hex) plus 20h

3.2 Header Request

Command ID Character: B

Format:

B..... .....Command ID Character
<mode> ....<mode> is either 20h to command the return of all header information, or 20h ORed with the parameter number (1-12 decimal expressed in hex or binary) to command the return of a particular parameter, or 30h ORed with the parameter number to command the field unit to modify the particular parameter from the remaining characters of the Header Request command.
<parameter data, if required>

Description:

The header of a field unit contains the parameters of the field unit as described below. These parameter may be collectively requested or may be individually requested using the header request command. Furthermore, the base station uses the header request command to change an individual parameter of the header by passing the new parameter value to the field unit in the command with a mode value of 30 hex ORed with the parameter to be modified.

Responses from Field Unit:

---

1. Fixed length data blocks contain data collected at the explicit request of the base station (immediate fixed length block or future fixed length block).

3.2.1 Response for Mode 20h Command

Format ID Character: D

Format:

	D	Format ID character
	unit #	Field unit ID number (0-3fh) plus 20h
[1[1]]	Time-of-day	
	tenths.....	a BCD nibble added to 20h
	seconds....	a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
	minutes	same as above
	hours.....	same as above
	days	days is 2 binary bytes (least significant first); converted to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978
[2]	Peak Threshold[2]	....2 byte positive signed binary number representing the peak acoustic level for which threshold events are to be taken. Convert to nibbles and then to 4 printable ASCII characters.
[3]	Wind Speed[3]	The wind speed threshold which, for winds averaging above this level, the WINDY flag will be set in the data blocks. A two digit decimal number expressed in BCD (two nibbles) computed knowing type of windmeter, desired threshold in MPH, and the wind sample interval.
[4]	Status	This is 2 bytes of data, the first of which is the same as one reads on the NEC portable terminal using "32H IN .HB". The interpretation of this byte can be found in the operations manual. This data is split into nibbles (least significant nibble first) and added to 20h to form 2 printable ASCII characters. The second byte is the high byte of the "error count" relating to the first-in-first-out buffer. A count greater than zero indicates a serious processing error within the field unit is highly likely.

1. The number in square brackets is the parameter number (in decimal) which is used to construct the MODE in order to read or set an individual parameter.

2. The peak threshold is a scaled value just as all other acoustic levels reported by or sent to the field unit. Thus, when reading the peak threshold from the header, add the calibration constant to obtain the true peak level in dB SPL. When setting the peak threshold, subtract the calibration constant before sending the value to the field unit.

3. See the explanation of the wind speed threshold, wind timeout, and wind sample interval in the revised operations manual.

[5] Wind Timeout[1].....8 bit binary number expressing the wind timeout in tenths of seconds; converted to two printable ASCII characters.

[6] Wind Sample Interval[1] ....
8 bit binary number expressing the wind sample interval in 1/20 th of seconds (0.05); converted to two printable ASCII characters

[7] Pre-trigger time[2]....4 - bit binary expressing the pre-trigger time in tenths of seconds, allowable range 1-7 decimal; converted to a printable ASCII character.

[8] Post-trigger time[2]..8 bit binary number expressing the post-trigger time in tenths of seconds, allowable range 1-255 decimal; converted to two printable ASCII characters.

[9] Wind Ignore. .....One character used as a flag, 20h to allow the field unit to use the "windy" flag to determine data stored, 21h to always store the data regardless of wind conditions.

[10] Autocalibration Interval...
4 bit binary in the range of 0-10 decimal. zero disables autocal, else the number represents the hours between autocals; converted to one printable ASCII character.

[11] Current Task[3] .....4 bit binary number, 0,1 or 2 decimal When read reveals the current field unit task; when written, sets the present task and the power-up or reset task (normally task 1); converted to one printable ASCII character.

[12] Phone Number.....arbitrary length ASCII string ending with a space, contains the base station telephone number in Hayes Smartmodem format.

Note that a special function to clear the field unit of collected data blocks uses an invalid parameter (namely 14 decimal) in a set header data command. Sending a Header Request with a mode of 3e hexadecimal wipes out all data blocks stored in the field unit.

---

1. See the explaination of the wind speed threshold, wind timeout, and wind sample interval in the revised operations manual.

2. See the explanation of pre-trigger and post-trigger in the revised operations manual.

3. The field unit has three primary modes or tasks; task 0 is STANDBY (unit will Autocal but take no threshold mode data), task 1 is THRESHOLD (data is recorded when the peak threshold is exceeded), and task 2 is FIXED LENGTH (unit is recording a fixed length block of acoustic data in response to an explicit command from the base station).

3.2.2 Response for Mode 2xh or 3xh Command

where x is the parameter number expressed as
a hexadecimal digit

Mode 2xh is used to read a particular parameter from the field unit.

Mode 3xh (followed by the required data) is used to write (or send) a parameter change to the field unit.

In each case, the field unit returns the following block; either providing information as to the present setting in case of mode 2xh, or an echo of the parameter value which has just been written or changed.

Format ID Character: D

Format:

D ....Format ID Character
unit # ....Field unit ID number(0-3fh) plus 20h
data .. ..the data for the particular parameter requested
in the format given in Section 3.2.1

3.3 Link Request[1]

Command ID Character: C

Format:

C ....Command ID Character
unit # ....Field unit ID number(0-3fh) plus 20h

Description:

This command initiates an exchange of dialogue between the base station and a particular field unit with the explicit unit #. Dialogue is terminated by either the field unit or the base station by transmitting "DLE".

Response from field unit:

1. Link request is not used in the dial-up mode; it is appropriate only for polled mode systems. In polled mode systems, each field unit is continually listening to the base station transmissions but never transmitting themselves. The link request allows the base station to request a particular field unit to "come on line" and begin two way communication.

3.3.1 Link Response

Format ID Character: G

Format:

G ....Format ID Character
unit # ....Field unit ID number (0-3fh) plus 20h

3.4 Listen Mode

Command ID Character: D

Format:

D ....Command ID Character
duration ....12 bit binary number in unit of seconds; three nibbles (least signification first) converted to 3 printable ASCII characters.

Description:

The purpose of listen mode is to allow the operator to listen to the actual microphone audio at the field unit with which he is in contact. This is useful for diagnosing problems with the microphone system or merely to satisfy ones curiosity if unexplainable data encountered.

Response from Field Unit:

The field unit will respond with an ACK is the message was understood, or NAK is an error occurred. The field unit will connect the microphone audio to the phone line and remove the modem carrier for the length of time requested. The field unit will hang-up the telephone at the end of the time period requested.

3.5 Immediate Fixed Length Block Request

Command ID Character: E

Format:

E..... ....Command ID Character
duration ....12 bit binary number in unit of seconds; three nibbles (least signification first) converted to 3 printable ASCII characters.
calibrator ....the sample may be taken with the B & K 4921 calibrator either on or off. 20h is sent to indicate no calibrator; 21h is sent to indicate the measurement should be taken with the calibrator on.

Description:

This command allows the operator to cause a sample of the acoustic data on demand. It may be used to collect data on the background levels and the field unit or to record a known low level event. It can also be useful as a diagnostic tool when the health of the system is suspected.

Response from Field Unit:

The field unit will respond with an ACK is the message was understood, or NAK is an error occurred. The data which the field unit will take in response to this command must be collected by the base station requesting data with a Type A command (Data Request).

3.6 Future Fixed Length Block Request

Command ID Character: F

Format:

F..... ....Command ID Character
Time-of-event
  tenths..... .....a BCD nibble added to 20h
  seconds.... .....a binary byte converted to 2 printable ASCII characters as above (least significant nibble first)
  minutes ....same as above
  hours..... .....same as above
  days ....days is 2 binary bytes (least significant first); convert to 4 printable ASCII character as above. Represents the date in terms of the number of days after January 1, 1978
duration..... .....12 bit binary number in unit of seconds; three nibbles (least signification first) converted to 3 printable ASCII characters.

Description:

Same as above, but the sample may be scheduled to occur at any time in the future. The operator should verify the accuracy of the field unit clock before requesting a future fixed length sample since resetting the unit clock may shift the actual time the sample is taken.

Response from Field Unit:

The field unit will respond with an ACK is the message was understood, or NAK if an error occurred. The data which the field unit will take in response to this command must be collected by the base station requesting data with a Type A command (Data Request).

3.7 Enable/Disable Threshold Data Acquisition

Command ID Character: G

Format:

G ....Command ID Character
flag ... .20h will disable acquisition of threshold data; 21h will enable acquisition of threshold data.

Description:

This command allows the operator to disable (or enable) the threshold data acquisition mode of the field unit at will. Even if threshold mode is disabled, fixed length blocks may still be requested by the base station, and if autocalibration has not been disabled using mode 3ah of the Header Request command (Type B), calibrations will still be taken.

4) **Interpretation of Data**

Several of the fields described in the above protocol have some significant processing and/or explanation in order for the data contained to be reasonably interpreted. These include Peak position, sample length, SEL data, RMS data, Peak Data, and those parameters related to the windmeter.

4.1) Peak Position and Sample Length

The peak position and sample length as returned by the field unit are sent as 12 printable ASCII characters or 6 bytes of binary integer data. One count represents a time interval of 50 microseconds. One must divide by 20000 to obtain the time in seconds.

Sample length in threshold blocks (Format A) includes the pre-trigger interval and post-trigger interval in the sample length and have a maximum resolution of 0.1 seconds (2000 counts). Most base station programs subtract the quantity [pre-trigger + post-trigger -0.1] from the sample length in order to display an event length which presents the actual time the acoustic signal was above threshold.

Sample length in fixed length blocks (Format H) does not include pre-and post-trigger interval in the reported sample length. Maximum resolution is 0.1 seconds as for threshold blocks.

The peak position is also 6 bytes as the sample length; it is measured from the start of the pre-trigger interval. One may wish to subtract the pretrigger interval for the same reasons given above. The maximum resolution of the peak position is one sample or 50 microseconds.

4.2) Interpretation of Calibration Data Blocks

It is essential to consider calibration data blocks first since all data must be scaled by a factor determined by the analysis of the calibration data.

4.2.1) RMS Data Field[1]

The following procedure will demonstrate the procedure to convert the data received from a calibration into dB. The RMS data field is sent as 16 printable ASCII characters which must be converted to 8 bytes of binary data. This number represents the sum of the squares of the acoustic samples (each 50 microseconds) over a period of 3.2 seconds (3.2 seconds times 20000 samples per second equals 64000 samples). Assume the RMS data in a Format B block is 0000 0000 2811 5BB0 hexadecimal.

0000 0000 2811 5BB0 = 672226224 decimal.

RMS value = √[(sum of squares)/(number of samples)]

RMS value = 102.486754022

$RMS_{\text{expressed in dB}} = 20 \log_{10}(\text{RMS value}) = 40.2 \text{ dB}$

4.2.2) Peak Data[1]

The peak data is simpler to convert since it is a single number, no summation; however, it is in 2's complement notation (7FFF is the most positive number and 8000 is the most negative number). Convert to a positive number before converting to dB. Assume the two byte peak data is 00B0 hex; this equals 176 decimal.

$\text{Peak value}_{\text{expressed in dB}} = 20 \log_{10}(\text{peak value})$

$\text{Peak value}_{\text{in dB}} = 44.9 \text{ dB}$

4.2.3) Determination of the Calibration Constant

The data from the field unit (converted to dB) is still in an format internal to the field unit which is essentially a voltmeter. All levels reported in data blocks must be scaled to standard acoustic levels to be meaningful.

---

1. The numbers used in these examples are extracted from the operations manual(page 4-19 of the September 1985 manual or page 4-32 of the May 1989 manual).

This is the legitimate purpose of the calibration block data. The B & K 4921 Microphone System is equipped with an "actuator" which produces a signal in the microphone equivalent to approximately 90 dB SPL. This 90 dB level is called the "actuator constant and is entered into the base station ID List for each field unit. As is evident from the calculation of section 4.2.1, the field unit reported an RMS level of 40.2 dB; however, the measured signal was really 90 dB. One must add 49.8 dB to the reported level to scale the result to the correct number. This number is the "calibration constant" and must be added to each reported SEL and peak value returned by the field unit.

Given the "actuator constant" of the 4921 and the RMS value returned by the calibration block, the "calibration constant" is derived as follows:

"calibration constant" = "actuator constant" - RMS value

4.3) Interpretation of SEL Data Blocks[1]

The processing of SEL data is similar to the processing of the RMS data except that the SEL is not an average value but an integral over time (a summation of samples in the discrete domain). The raw data is the square of the samples summed over the duration of the event, each sample representing a 50 microsecond period. Since the units for SEL are dB-seconds, a correction factor is applied to correct for the 50 microsecond sampling period.

SEL value = [sum-of-squares]/[number of samples per second}

The SEL is then converted to dB representation.

$$SEL_{\text{expressed in dB}} = 10 \log_{10}(\text{SEL value})$$

If the binary SEL value is 0000 0DCD 2180 expressed in hexadecimal (231,547,264 decimal), then

SEL value = 231547264/20000 = 11577.3632

$$SEL_{\text{in dB}} = 40.6 \text{ dB}$$

Note that the calibration constant must be added to this number to make the level reported relate to 0 dB = = 20 microPascals.

1. Blocks of Format A or Format H.

4.4) Interpretation of Windmeter Parameters

The wind speed timeout and the wind sample interval are binary integers with a resolution of a tenth of a second. For the system to function properly, the wind speed timeout must be greater than the wind sample interval.

The relationship among the wind sample interval, the 2-digit BCD number called "wind speed" but really a COUNT, and the desired wind speed threshold will be explained in the following paragraphs.

The wind measuring system of the field unit is a counter which is preset to a starting count each wind sample interval period. Between presets, the counter is being clocked by the pulses generated by the anemometer. Each pulse decrements the counter one count. If the counter reaches zero, a hardware interrupt is generated; the occasion of this hardware interrupt is the signal that wind speed threshold has been exceeded.

The anemometer produces pulses whose frequency of occurrence increases with wind speed and the higher the starting count preset into the counter, the more anemometer pulses will be required to decrement it to zero. This translates to wind speed in miles per hour depending on how long the anemometer has to decrement the counter between presets.

WSI = wind sample interval in seconds

MPH = desired wind speed threshold in miles/hour

COUNT = the 2-digit decimal number representing the starting count to be preset

For the W203 windmeter (most installation except APG:

$$COUNT = floor\{[5.1*MPH - 6.0]*WSI + 0.5\}$$

or

$$MPH = floor\{[(COUNT/WSI) + 6]/5.1 + 0.5\}$$

where floor(x) is the largest integer less than or equal to x.

Distribution

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: HECSA Mailroom (2)
ATTN: CECC-R
ATTN: CERD-L
ATTN: CERD-M

Fort Drum 13602-5097
ATTN: ATZS-PW-E (10)

Defense Tech Info Center 22304
ATTN: DTIC-O (2)

19
12/99

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1999	Final

4. TITLE AND SUBTITLE
CERL Noise Monitoring and Warning System 98

5. FUNDING NUMBERS
MIPR
6MCER50063 H16

6. AUTHOR(S)
Daniel Sachs, Jonathan W. Benson, and Paul D. Schomer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Construction Engineering Research Laboratory (CERL)
P.O. Box 9005
Champaign, IL 61826-9005

8. PEFORMING ORGANIZATION REPORT NUMBER
TR 99/99

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Commander, Fort Drum
ATTN: ATZS-PW-E
Bldg. 4838
Fort Drum, NY 13602-5097

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

9. SUPPLEMENTARY NOTES
Copies are available from the National Technical Information Service, 5385 Port Royal Road, Springfield, VA 22161

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The present CERL Noise Monitoring and Warning System, designed in the mid-1980s, has difficulty separating blast noise sounds from wind-induced pseudo-noise. A new noise monitor was designed that would be more wind noise resistant and would use more modern electronics and methods than those available in 1985. This report documents the design and construction of this new noise monitor.

The heart of wind-noise resistance is a two-microphone array and special signal processing to identify and separate blast sounds from pseudo-wind noise. The results are quite encouraging. It appears that the new monitor improves the signal-to-noise ratio by about 10 dB. It is recommended that this monitor be transferred to the field by a demonstration validation program such as the Environmental Security Technology Certification Program (ESTCP).

14. SUBJECT TERMS
noise
noise measurement
warning system
blast noise
ESTCP

15. NUMBER OF PAGES
184

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ASTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR